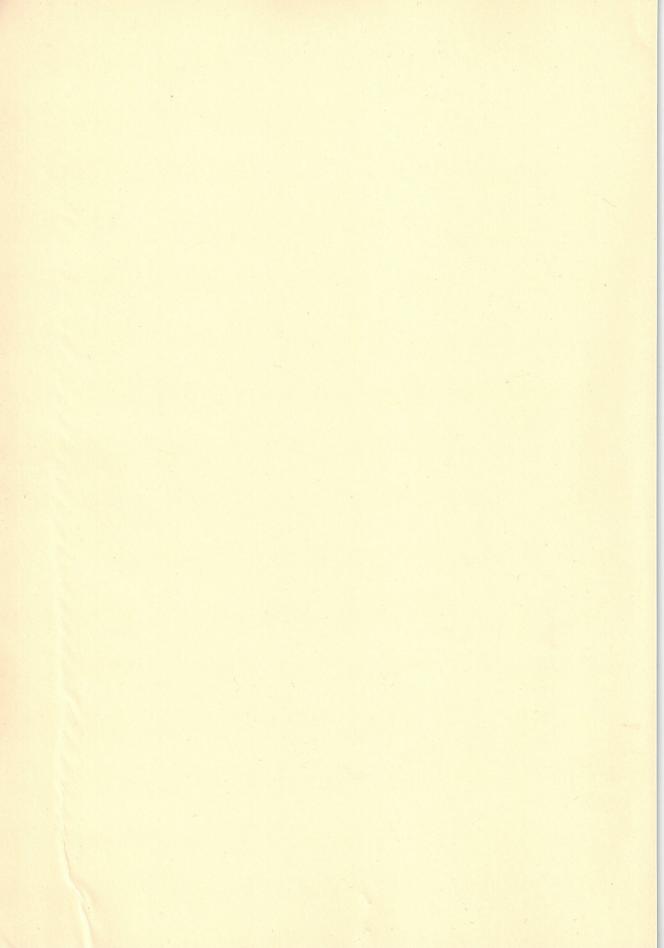
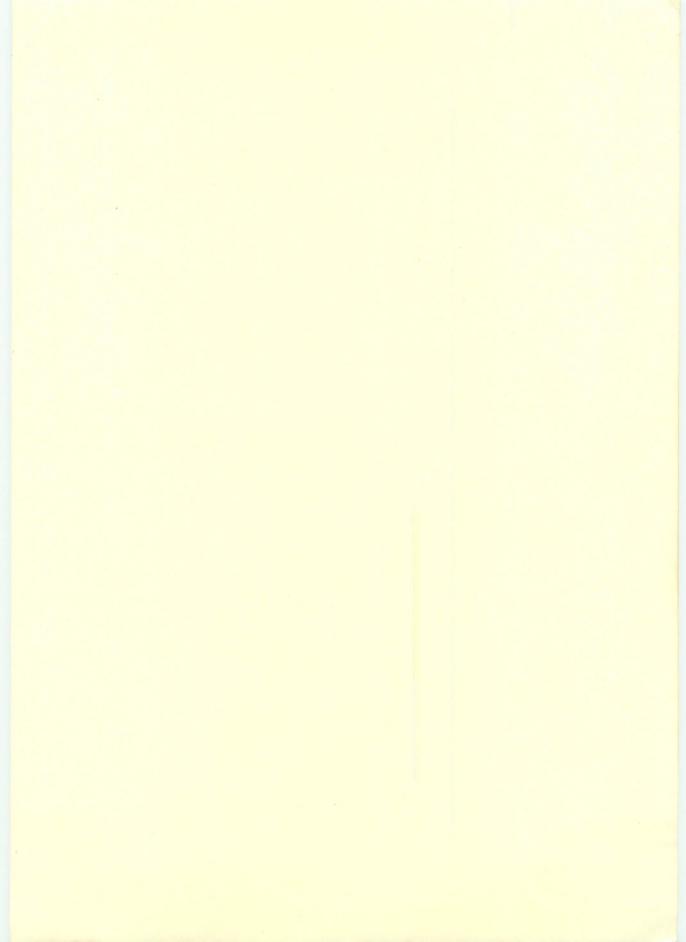


Z8000 Data Manual











Z8000 Data Manual

Printed in Japan

28000 Data Manual

Polis

Copyright © 1980 by Zilog, Inc. All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior written permission of Zilog.

Zilog assumes no responsibility for the use of any circuitry other than circuitry embodied in a Zilog product. No other circuit patent licenses are implied.

The information contained herein is subject to change without notice.

Z8000 Data Manual Contents

ソフトウエア編

| Z8000 PLZ/ASM Assembly Language Programming Manual |
|---|
| Introduction to RIO Text Processing |
| Z8000 PLZ/ASM Assembler User Guide |
| Z8000 Linker User Guide |
| MacP User Guide |
| Z80/Z8000 Translator User Guide |
| Z8000 Development Module Timing and Logic Diagram |
| Z8000 Development Module Monitor Program Listing |
| |
| ハードウエア編 |
| Z8001/Z8002 CPU Product Specification |
| Z8010 MMU, Memory Management Unit Product Specification |
| A Small Z8000 System, Application Note |
| Zilog Z8000 Family, Technical Overview |

(注;本データ・マニュアルは上記個別のデータ・ブックを集録したものです)

Z2000 Data Manual Contents

Z8000 PLZ/ASM Assembly Language Programming Manual

Preface

This manual describes the PLZ/ASM assembly language for the Zilog Z8000 microprocessor, and serves as the primary reference manual for the assembly language programmer. It is one in a series of documents describing the Z8000 and associated hardware and software. Information on the use of the PLZ/ASM Assembler can be found in the publication:

Z8000 Assembler User's Guide

It is intended that this manual be read the first time in sequence, from beginning to end.

Chapter 1 provides an overview of the architecture of the Z8000, with special emphasis on those features of interest to the assembly language programmer. A detailed description of the architecture and hardware-related features can be found in the publication:

Z8000 CPU Technical Manual (To be published)

Chapter 2 introduces assembly language conventions, including the format of statements and addressing mode specifications.

Chapter 3 contains the detailed instruction set of the Z8000.

Chapter 4 introduces the high-level PLZ/ASM statements needed to construct a complete program, while Chapter 5 contains the detailed description of these statements.

The Appendices contain a summary of the instruction set, high-level statements and assembler directives, plus a table of the ASCII character set.

Contents

| SECTION | 1. | ARCH | ITECTURA | AL OVERVIEW | 12 |
|---------|----|-------|--|--|----------------------------------|
| | | 1.1 | INTRODU | JCTION | 12 |
| | | 1.2 | MEMORY | ADDRESS SPACES | 14 |
| | | | 1.2.1 | Four Memory Address Spaces Addressing Memory Spaces | 16 16 |
| | | 1.3 | INPUT/ | OUTPUT ADDRESS SPACES | 17 |
| | | 1.4 | SEGMENT | PATION | 19 |
| | | | 1.4.1 | Nonsegmented and Segmented Addresses Memory Management | 20 |
| | | 1.5 | DATA MA | ANIPULATION | 26 |
| | | | 1.5.1 1.5.2 1.5.3 1.5.4 1.5.5 | Data Types | 26 27 27 29 35 |
| | | 1.6 | PROGRAM | M CONTROLS | 35 |
| | | | 1.6.1 1.6.2 1.6.3 1.6.4 1.6.5 1.6.6 | Program Counter Status Flags Control Bits Interrupts and Traps Program Status Area System Reset Memory Refresh | 37 39 39 41 44 44 |
| | | 1.7 | ADDRESS | ARITHMETIC | 46 |
| | | | 1.7.1 | Nonsegmented Addressing Segmented Addressing | 46 46 |
| SECTION | 2. | Z8000 |) ASSEME | BLER CONVENTIONS | 48 |
| | | 2 7 | ACCEMBI | ED OVERVIEW | |

| | 2.2 | ASSEMBLY LANGUAGE STATEMENT FORMAT 49 | |
|-----------|----------|---|-------|
| | | 2.2.1 Program Labels and Identifiers 49 2.2.2 Instruction | |
| | 2.3 | ARITHMETIC OPERANDS 53 | |
| | | 2.3.1 Run-Time vs. Assembly-Time Arithmetic 53 | |
| | | 2.3.2 Constants 54 2.3.3 Data Variables 55 2.3.4 Expressions and Operators 57 2.3.5 Segmented Address Operators 60 | |
| | 2.4 | Z8000 ADDRESSING MODES | |
| | | 2.4.1 Immediate Data 62 2.4.2 Register Address 63 2.4.3 Indirect Register Address 64 2.4.4 Direct Address 65 2.4.5 Indexed Address 66 2.4.6 Relative Address 67 2.4.7 Based Address 68 2.4.8 Based Indexed Address 69 | |
| SECTION 3 | B. ASSE | MBLY LANGUAGE INSTRUCTION SET 70 | |
| | 3.1 | FUNCTIONAL SUMMARY 70 | |
| | 3.2 | NOTATION AND BINARY ENCODING 80 | |
| | | 3.2.1 Operand Notation | |
| | 3.3 | ASSEMBLY LANGUAGE INSTRUCTIONS 92 | |
| | 3.4 | UNIMPLEMENTED INSTRUCTIONS 254 | |
| SECTION 4 | 4. STRUC | CTURING A Z8000 PROGRAM255 | N. P. |
| | 4.1 | TNTRODUCTION SIVER OVERVIEW NOITCHORTHI | |

| Cookiel | 4.2 | PROGRAM STRUCTURE 2 | 55 |
|------------|------|---|--|
| | | 4.2.2 Procedures 2 4.2.3 DO Loops 2 4.2.4 IF Statements 2 4.2.5 Scope 2 | 255 256 257 258 259 260 |
| | 4.3 | RELOCATABILITY 2 | 262 |
| | | 4.3.2 Location Counter Control 2 | 262 263 264 |
| SECTION 5. | PLZ/ | ASM HIGH-LEVEL STATEMENTS 2 | 267 |
| | 5.1 | Z8000 SOURCE PROGRAM STATEMENTS 2 | 267 |
| | 5.2 | PROGRAM STRUCTURING STATEMENTS 2 | 868 |
| | | 5.2.2 Procedure Declaration | 268 268 270 271 273 |
| | 5.3 | DEFINING DATA 2 | 274 |
| | | 5.3.2 Data Types | 274 275 278 280 285 286 |
| APPENDIX A | HIGH | I-LEVEL STATEMENTS SUMMARY | |
| APPENDIX B | ASSE | MBLER DIRECTIVES AND | |
| | | ASSEMBLER DIRECTIVES | |

| | B.2 EXTENDED INSTRUCTIONS300 |
|-------------|--|
| APPENDIX C | RESERVED WORDS AND SPECIAL CHARACTERS |
| | |
| | |
| | LIST OF TABLES |
| TABLE 3-1 | NUMBER OF BYTES IN INSTRUCTIONS 89 |
| | LIST OF ILLUSTRATIONS |
| FIGURE 1-1 | Z8000 PIN Functions |
| FIGURE 1-2 | Addressable Data Elements18 |
| FIGURE 1-3 | Segmented Address (Register Memory)21 |
| FIGURE 1-4 | Segmented Address Within Instruction21 |
| FIGURE 1-5 | The MMU Connection23 |
| FIGURE 1-6 | Logical to Physical Address Translation24 |
| FIGURE 1-7 | Segmented Address Relocation |
| FIGURE 1-8 | Byte and Word Strings |
| FIGURE 1-9 | General Purpose Registers30 |
| FIGURE 1-10 | Typical Instruction Formats (nonsegmented) 36 |
| FIGURE 1-11 | Program Status Blocks |
| FIGURE 1-12 | Format of Saved Program Status in the System Stack |

| | | | | J 11 | | | | |
|--------|------|---------|---------|------|-------|----|---------------|----------|
| FIGURE | 1-13 | Program | Status | Area | | | | . 43 |
| FIGURE | 1-14 | Program | Status | Area | Point | er | | . 45 |
| FIGURE | 1-15 | Refresh | Registe | er | | | • • • • • | . 45 |
| | | | | | | | | |
| | | | | | | | | |
| | | | | | | | | |
| | | | | | | | | |
| | | | | | | | | |
| | | | | | | | | |
| | | | | | | | | |

Section 1 Architectural Overview

1.1 INTRODUCTION

Zilog's Z8000 microprocessor has been designed to accommodate a wide range of applications, from the relatively simple to the large and complex. Depending on the Z8000 configuration chosen, the programmer can directly address from 64 kilobytes (64K or 65,536 bytes) to 8 megabytes (8M or 8,388,608 bytes) of memory.

The Z8000 achieves high throughput with a relatively low clock rate (standard: 4MHz) and can, therefore, use memories with a comparatively long access time. Built-in random-access memory (RAM) refresh with a programmable refresh rate permits the use of a wide variety of dynamic memories.

Z8000 central processing unit (CPU) resources include sixteen 16-bit general-purpose registers, seven data "types" (lengths from single bits to 32-bit long words), eight addressing modes, and a repertoire of 105 instructions. These resources are similar in form to comparable features of the Z8-microcomputer and Z80-microprocessor families, allowing users of these systems to upgrade easily to the Z8000.

Over 410 meaningful combinations of instructions, data types, and addressing modes are available with the Z8000. The instruction set also includes signed multiplication and signed division (implemented in hardware) for both 16-bit and 32-bit values.

The Z8000 provides several sophisticated features not found on other microprocessors. Since reference to these features are made throughout this manual, a brief description of each of the major concepts and terminology follows:

Segmentation

To facilitate the management of a large address space, the Z8000 allows program and data to be accessed as part of variable-length logical groupings called "segments". Segmentation provides hardware assistance for relocation of program and data, as well as protection against accidental or malicious damage to system or user information (see Section 1.4).

• Stacks

The Z8000 has several instructions which facilitate the handling of stacks (Section 1.5.3). A "stack" is an area of memory used for temporary storage or re-entrant procedure call/interrupt service linkage information, and is managed in a "last-in, first-out" manner; that is, several items may be "pushed" or added to the top of a stack and then "popped" or removed from the stack in reverse order. On the Z8000, a stack starts at the highest address allocated for it and grows linearly downward to the lowest address as items are pushed. A general-purpose register is used to "point" to the current top of the stack; that is, the register contains the address of the most recently-pushed item.

• Interrupts and Traps

Two events can alter the normal execution of a Z8000 program: asynchronous hardware "interrupts" which occur when a peripheral device needs service, and synchronous software "traps" which occur when an error condition arises such as an attempted use of an unimplemented instruction. The Z8000 handles both of these cases in a similar manner (Section 1.6.4). The current status of the processor is pushed on a stack, and then program control is automatically "vectored" to a handler procedure for the particular class of interrupt or trap. The address of the handler is determined by indexing a table (or "vector") of program status blocks which is referred to as the Program Status Area, and extracting the appropriate entry that includes the address of the handler procedure. When the handler procedure is finished, control can be returned to the interrupted program through the status information saved on the stack. (Non-vectored interrupts are also possible, with the burden placed on the program to determine which device needs servicing.)

Normal and System Operating Modes

Z8000 programs run in one of two operating modes:

Normal mode or System mode. Normal mode is the operating mode during normal program execution. System mode exists primarily to protect the operating system kernel. Instructions that alter the machine state (such as I/O operations, changes to control registers, etc.) can be issued only in System mode and are referred to as "privileged" instructions.

Attempting to execute privileged System mode instructions while in Normal mode initiates a trap. This is an important feature in a multi-user environment. Programs running in normal mode cannot take over or monopolize control of the Z8000 or damage other users' programs or system software. System mode is also in effect following an interrupt or trap.

Compiler and assembler code, the program code output by these translators, and operating system code, all run efficiently on the Z8000. Large address spaces, memory relocation, system and normal stacks, special instructions, and a sophisticated interrupt and trap structure add to this efficiency.

The Z8000 can also be part of a multi-microprocessor system using its exclusion and synchronization software instructions and its Micro Input and Micro Output hardware controls. The large address space of the Z8000 augments the overall data-processing capability of multi-microprocessor configurations.

From a programmer's point of view, the basic Z8000 system configuration consists of the Z8000 microprocessor and its various memory and I/O address spaces. These are described in the remainder of this section. For detailed architectural and configuration data, see the Z8000 Technical Manual.

NOTE

The Z8000 is designed to provide for future extensions to its architecture which would require several of the currently unused (but reserved) bits in some of the instruction and data encodings. Any shaded or hatched areas in the figures throughout this manual are considered reserved and should not be used by the programmer. Reserved bits must be zero.

1.2 MEMORY ADDRESS SPACES

To allow for a wide range of applications, two versions of the Z8000 microprocessor device are available: a 40-pin package (Z8002) with an address range of 0-64K bytes, and a 48-pin package (Z8001) with an address range of 0-8M bytes (Figure 1-1). The Z8002 version is called nonsegmented and the Z8001 is the segmented version (see Section 1.4.1).

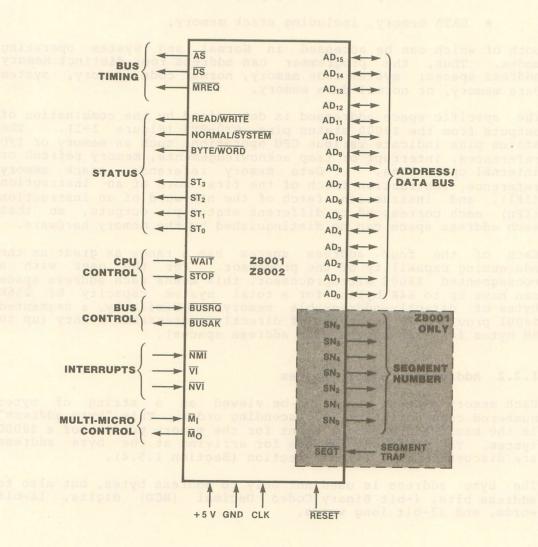


Figure 1-1 Z8000 Pin Functions

1.2.1 Four Memory Address Spaces

When memory is addressed, the Z8000 CPU distinguishes between

- CODE memory, containing program instructions, and
- DATA memory, including stack memory,

both of which can be accessed in Normal and System operating modes. Thus, the programmer can address four distinct memory address spaces: system code memory, normal code memory, system data memory, or normal data memory.

The specific space addressed is determined by the combination of outputs from the Z8000 status pins, ST0-ST3 (Figure 1-1). The status pins indicate various CPU operations such as memory or I/O references, interrupt or trap acknowledgements, memory refresh or internal operations. Data memory reference, stack memory reference, instruction fetch of the first word of an instruction (IF1), and instruction fetch of the nth word of an instruction (IFn) each correspond to different status pin outputs, so that each address space can be distinguished by the memory hardware.

Each of the four address spaces has a range as great as the addressing capability of the processor. For the user with a nonsegmented Z8002 microprocessor, this means each address space can have up to 64K bytes, for a total system capacity of 256K bytes of directly-addressable memory. Similarly, a segmented Z8001 provides 32 megabytes of directly-addressable memory (up to 8M bytes for each of the four address spaces).

1.2.2 Addressing Memory Spaces

Each memory address space can be viewed as a string of bytes numbered consecutively in ascending order. This "byte address" is the basic addressing element for the memory portion of a Z8000 system. The different methods for arriving at the byte address are discussed later in this section (Section 1.5.4).

The byte address is used not only to address bytes, but also to address bits, 4-bit Binary Coded Decimal (BCD) digits, 16-bit words, and 32-bit long words.

In the case of a bit or digit, the memory address designates the byte or word that contains it. For example, a bit can be addressed by specifying a byte address and the number of the bit within the byte (0-7), or by specifying a word address and the bit number within the word (0-15). Bits are numbered right-to-left; that is, least-to-most significant (Figure 1-2).

In the case of data types longer than one byte, the memory address designates the leftmost, or high-order byte. In other words, the high-order byte has the lowest memory address of the bytes within a word or long word (Figure 1-2).

Program instructions residing in code memory are always addressed as 16-bit words. Words and instructions are always aligned (the high-order byte must have an even-numbered address). Aligned words improve access speed and double the range of instructions that use relative addresses (JR, DJNZ, DBJNZ, CALR) -- see Section 1.5.4. The memory address of a long word must also be an even-numbered byte address.

NOTE

Word quadruples (64 bits) cannot be addressed in memory. It is possible to address 64-bit register quadruples, however. This is covered later, in Section 1.5.2.

1.3 INPUT/OUTPUT ADDRESS SPACES

The Z8000 has an I/O address space separate from the memory address space. I/O address references can be distinguished from memory addresses using the status pin outputs from the Z8000 (Figure 1-1).

I/O addresses are 16-bit addresses allowing a range of 0-64K bytes to be addressed. I/O addresses are multiplexed with the byte or word data accessed by the I/O operation. (See pins ADO - AD15 of Figure 1-1.) I/O data references have an automatic wait cycle included for each load or store, which results in a 4-cycle read or write.

The Z8000 instruction set includes two groups of I/O instructions: a standard complement of input, output, and block transfer instructions, plus a "special" group of I/O instructions. The latter are generally used to load and examine the Memory Management Unit, a device used only in the segmented Z8000 system configuration (Section 1.4.2). Special I/O references can be distinguished from standard I/O references using the status pins (Figure 1-1.)

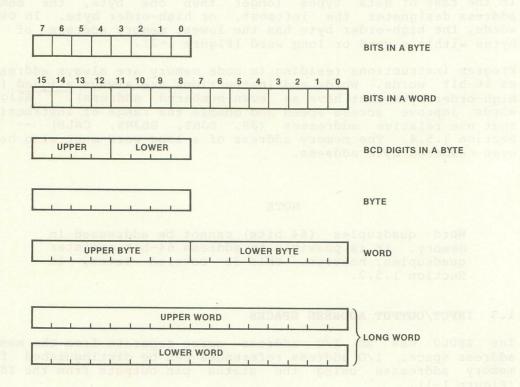


Figure 1-2 Addressable Data Elements

1.4 SEGMENTATION

The segmented Z8000 memory address space may be separated into several (up to 128) variable-length "segments". Each segment may vary in size, independently from other segments, from 0 to 64K bytes. Addresses consist of both a segment number and an offset. The segment number is used as an index into a table of base addresses for each segment. The table is kept in a separate package called the Memory Management Unit (MMU). The corresponding table entry provides a base to which the offset is added, thus providing the final address. The term "logical address" is used to indicate the segmented address used by the programmer to construct his program. The term "physical address" is used to indicate the translated address which is passed to the physical memory hardware. This address translation is accomplished completely in hardware and does not affect the instruction execution time.

Segmentation provides:

- 1) Dynamic relocation of program and data memory without necessitating the modification of addresses. Since all instructions use addresses which are relative to the segment base registers, code and data segments can be relocated anywhere in memory simply by moving the information and setting the segment base registers (in the MMU) to new values. This provides great flexibility for multi-user or multi-task operating systems in efficiently managing memory allocations.
- 2) Hardware-assisted memory protection to insure that only valid addresses within the bounds of the user's code and data segments are accessed. Since segments can be of variable length, only the amount of memory needed for each segment needs to be specified (independently from other segments).
- 3) Large logical address space without necessitating large physical address space ("virtual memory"). Since logical addresses are 23 bits, up to 8M bytes per address space can be accessed. This space usually exceeds the size of the available physical memory address space. However, by using a combination of the MMU, some external hardware and some operating system software, a virtual memory scheme can be employed which detects accesses to logical memory which are not currently mapped to physical memory, thus allowing the loading of segments from external secondary memory devices on demand.
- 4) Controlled sharing of memory with several Z8000 CPUs. Logical segments may be shared by several microprocessors by using the same MMU, or the same physical memory segments may be shared with several MMUs mapping different microprocessors'

logical address spaces into the same physical addresses. Relocation, protection and/or virtual memory facilities may be shared by several processors as well.

The nonsegmented Z8002 uses 16-bit addresses that can be manipulated as words. This version can directly address 64K bytes of memory in each of its four address spaces. The segmented, Z8001 version uses 23 bits to address directly up to 8M bytes in each of its four address spaces. The basic difference between programs running in nonsegmented or segmented modes is the number of bytes used to form addresses in instructions or registers.

Code written for a nonsegmented Z8000 can run in one segment of a segmented Z8000. This is called "running the segmented Z8000 in nonsegmented mode". The converse is not possible; i.e., code written for a segmented Z8000 will not run properly on a nonsegmented Z8000 due to differences in the instruction formats, and the use of register pairs for addressing modes. The functionality of the two versions is identical in all respects other than memory addressing.

1.4.1 Nonsegmented and Segmented Addresses

All nonsegmented addresses are represented as 16-bit words, whether they reside in a register, in memory, or as part of an instruction.

Segmented addresses require 23 bits. Each 8-megabyte address space is divided into 128 segments from 0 to $64 \, \mathrm{K}$ bytes each. Thus, to address a byte in one of these spaces, two pieces of information are needed:

- the <u>segment number</u> (0-127), which can be expressed in 7 bits, and
- the offset from the beginning of the segment (i.e., the 0-64K byte address within the segment), which can be expressed in 16 bits.

The two parts of a segmented address may be manipulated separately. Word functions, including 16-bit arithmetic, can be performed on the offset portion.

Figures 1-3 and 1-4 show internal representations of segmented addresses. The two words shown in Figure 1-3 could represent a long word in memory (Section 1.2.2) or a register pair in the CPU (Section 1.5.2). Within instructions, segmented addresses can have a long 16-bit offset or a short 8-bit offset. As Figure 1-4 indicates, bit 7 of the segment-number byte is used to differentiate between these two formats.

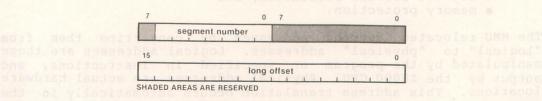
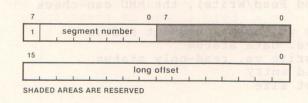


Figure 1-3 Segmented Address (Register Memory)



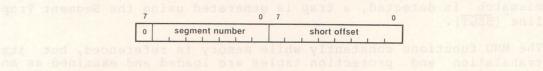


Figure 1-4 Segmented Address Within Instruction

1.4.2 Memory Management

The segmented Z8000 configuration usually includes a separate package called the Memory Management Unit (MMU). The MMU pin functions are pictured logically in Figure 1-5 and described below. These functions are divided between:

- segment address relocation, and
- · memory protection.

The MMU relocates segment addresses by converting them from "logical" to "physical" addresses. Logical addresses are those manipulated by the program or specified in instructions, and output by the Z8000 CPU. Physical addresses are actual hardware locations. This address translation occurs automatically in the MMU and requires no programmer intervention (see Figure 1-6).

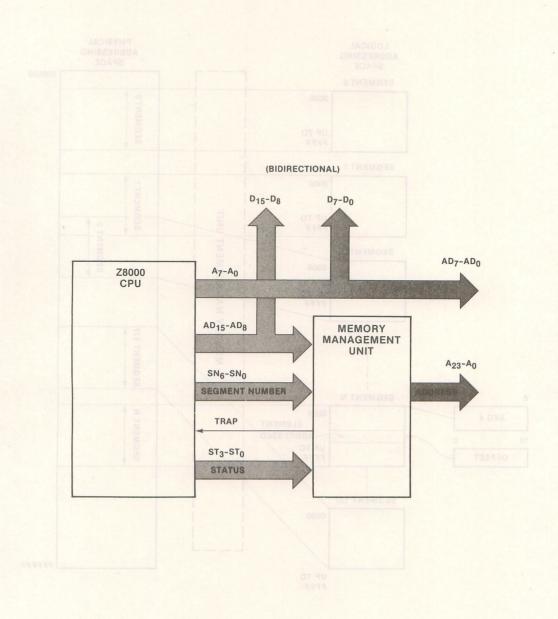
As Figure 1-5 illustrates, the inputs to the MMU include the segment number and the upper eight bits of the offset. Each segment number is associated with a 24-bit "base" address equivalent to the first physical address in the segment. The 16-bit offset is added to this base to complete the 24-bit physical address. Note that the lower eight bits of the offset are passed directly to the physical memory, so that the MMU need only store the upper 16 bits of each base address (bits 8-23). Figure 1-7 is another representation of the addition done by the MMU to form the 24-bit physical address.

The other major function of the MMU is memory protection. By interpreting its four status lines (Chip Select, Address Strobe, Data Strobe, and Read/Write), the MMU can check

- System vs. normal segment status
- · Code vs. data status
- Read/write vs. read-only status
- Invalid entry
- Segment size

During each memory reference, the attributes for each segment are checked against the corresponding Z8000 status lines. If a mismatch is detected, a trap is generated using the Segment Trap line ($\overline{\text{SEGT}}$).

The MMU functions constantly while memory is referenced, but its translation and protection tables are loaded and examined as an I/O peripheral. The Z8000's special I/O instructions (SIN, SOUT, and their variations — see Section 3) can load or examine the MMU.



golds least Figure 1-5 The MMU Connection - 1 symple

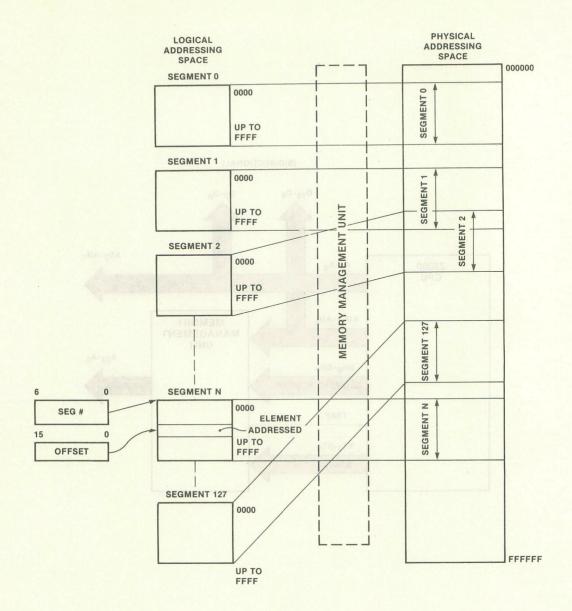


Figure 1-6 Logical to Physical Address Translation

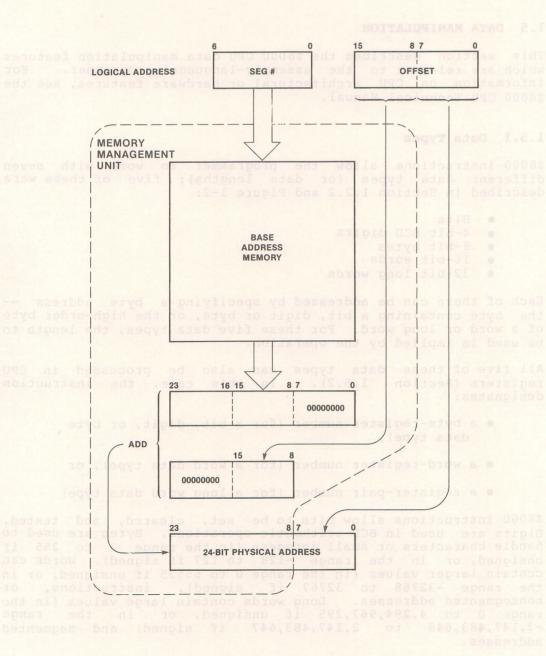


Figure 1-7 Segmented Address Relocation

1.5 DATA MANIPHLATION

This section describes the Z8000 CPU data manipulation features which are relevant to the assembly-language programmer. For information on CPU architectural or hardware features, see the Z8000 CPU Technical Manual.

1.5.1 Data Types

Z8000 instructions allow the programmer to work with seven different data types (or data lengths); five of these were described in Section 1.2.2 and Figure 1-2:

- Bits
- 4-bit BCD digits
- 8-bit bytes
- 16-bit words
- 32-bit long words

Each of these can be addressed by specifying a byte address — the byte containing a bit, digit or byte, or the high-order byte of a word or long word. For these five data types, the length to be used is implied by the operation.

All five of these data types can also be processed in CPU registers (Section 1.5.2). In this case, the instruction designates:

- a byte-register number (for a bit, digit, or byte data type),
- a word-register number (for a word data type), or
- a register-pair number (for a long word data type)

Z8000 instructions allow bits to be set, cleared, and tested. Digits are used in BCD arithmetic operations. Bytes are used to handle characters or small integers (in the range 0 to 255 if unsigned, or in the range -128 to 127 if signed). Words can contain larger values (in the range 0 to 65535 if unsigned, or in the range -32768 to 32767 if signed), instructions, or nonsegmented addresses. Long words contain large values (in the range 0 to 4,294,967,295 if unsigned, or in the range -2,147,483,648 to 2,147,483,647 if signed) and segmented addresses.

The two remaining data types, not discussed thus far are:

- · Strings of bytes
- · Strings of words

Strings can be stored and referenced only in memory. They are referenced by designating either their lowest or their highest byte address plus their length in bytes or words. They are used with Z8000 "autoincrement" or "autodecrement" instructions, which automatically increase or decrease the pointer to a byte or word address (Figure 1-8). Note that when strings are used, the programmer must designate the length of the data element explicitly, as one of the instruction operands.

1.5.2 General-Purpose Registers

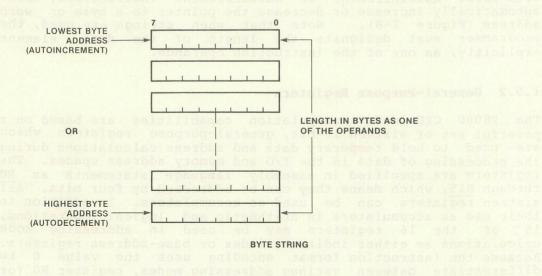
The Z8000 CPU data manipulation capabilities are based on a powerful set of sixteen 16-bit, general-purpose registers which are used to hold temporary data and address calculations during the processing of data in the I/O and memory address spaces. The registers are specified in assembly language statements as $\underline{\text{R0}}$ through $\underline{\text{R15}}$, which means they can be addressed by four bits. All sixteen registers can be used as accumulators. In addition to their use as accumulators in arithmetic and logical operations, 15 of the 16 registers may be used in addressing mode calculations as either indirect, index or base-address registers. Because the instruction format encoding uses the value 0 to differentiate between various addressing modes, register R0 (or the register pair RR0) cannot be used as an indirect, index or base-address register.

The programmer can also address registers as groups of 8, 32, or even 64 bits (Figure 1-9). These registers are specified using the following assembly language symbols:

- RHO, RLO, RHI, RLI, ..., RH7, RL7 for 8-bit registers. ("H" stands for high-order byte, and "L" stands for low-order byte within a word register). These registers overlap 16-bit registers RO - R7. All 8-bit registers can be used as accumulators.
- RRO, RR2, RR4, ..., RR14 for 32-bit register pairs.
- RQ0, RQ4, RQ8, and RQ12 for 64-bit register quadruples.
 These registers are used only by a few instructions such as Multiply, Divide, and Extend Sign.

1.5.3 Stacks and Stack Pointers

The Z8000 is a register-oriented machine. It also has sophisticated stack-oriented instructions and includes separate hardware-maintained stacks for its two operating modes -- a system stack and a normal stack, residing in system data memory and normal data memory, respectively. In addition, since any general-purpose register (except R0) can be used as a



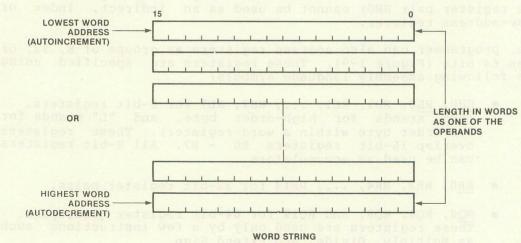


Figure 1-8 Byte and Word Strings

programmer-maintained stack pointer with instructions such as Push and Pop, multiple stacks can be efficiently supported. The stack pointers (pointers to stack locations in data memory) are kept in general-purpose registers and, therefore, can be altered using standard instructions. No special stack-pointer instructions are needed.

The <u>processor stack</u> is used implicitly by certain operations such as the Call and Return instructions or the interrupt and trap mechanisms. Depending on the System or Normal mode, these operations use a hardware-defined stack pointer to access either the system processor stack or the normal processor stack.

For the nonsegmented Z8000, the system and normal processor stacks can be addressed by 16 bits. Register R15 is used for this purpose. For the segmented version, 32-bit pointers are required and register pair RR14 is used (Figure 1-9). The format of the 32-bit address is shown in Figure 1-3.

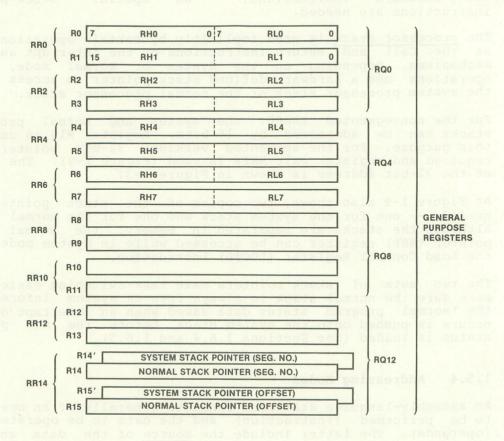
As Figure 1-9 also shows, two copies of the stack pointer are needed -- one for the system stack and one for the normal stack. Although the stacks are separated in memory, the normal stack pointer (NSP) register can be accessed while in System mode using the Load Control Register (LDCTL) instruction.

The two sets of stack pointers make task-switching easier. To make sure the normal stack is always free of system information, the normal program status data saved when an interrupt or trap occurs is pushed onto the system stack before the new program status is loaded (see Sections 1.6.4 and 1.6.5).

1.5.4 Addressing Modes

An assembly-language statement consists generally of an operation to be performed (instruction) and the data to be operated upon (operands). The latter include the <u>source</u> of the data and the <u>destination</u> where the result of the <u>operation</u> is to be stored.

In its simplest form, an operand can be either the specific data to be processed (immediate data) or the name of a register that holds the data. Data can be specified in more complex ways, also. For example, an operand may name a register whose contents are added to the contents of another register to form the address of the memory location containing the source data (based indexed addressing). Data can be specified by eight distinct addressing modes:



NOTE: Register R14 is not used as a stack pointer in the nonsegmented Z8000.

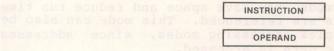
Figure 1-9 General-Purpose Registers

- Immediate Data (IM) prisasibh refeires destros
- Register (R)
- Indirect Register (IR)
- Direct Address (DA)
 - Indexed Address (X)
 - Relative Address (RA)
 - Based Address (BA)
 - Based Indexed Address (BX)

Depending on the operation, the addressing mode to be used might be implied by an instruction or spelled out explicitly. When stated explicitly, the addressing mode usually refers to a register or memory location. Implied addressing modes usually refer to program (code) memory or the I/O address space.

Immediate Data Black Made and algorithm and being bein

Although considered an "addressing mode" for the purpose of this discussion, Immediate Data is the only mode that does not indicate a register or memory address. The data processed by the instruction in this case is the value supplied as the operand.



THE OPERAND VALUE IS IN THE INSTRUCTION

Immediate Data mode is often used to initialize registers. The Z8000 is optimized for this function, providing several short immediate data instructions to reduce the byte count of programs.

Register Addressing

In Register addressing mode, the instruction processes data taken from a specified general-purpose register. Storing data in a register allows shorter instructions and faster execution.

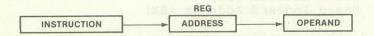


THE OPERAND VALUE IS THE CONTENTS OF THE REGISTER

The register length (byte, word, register pair, or register quadruple) is implied by the instruction.

Indirect Register Addressing

In Indirect Register addressing mode, the data processed is not the value in the specified register. Instead, the register holds the address of the data.



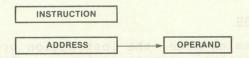
THE OPERAND VALUE IS THE CONTENTS OF THE LOCATION WHOSE ADDRESS IS IN THE REGISTER

A single word register is used to hold the address in Nonsegmented mode, while a register pair must be used in Segmented mode. Any general-purpose word register (or register pair in Segmented mode) can be used except RO or RRO. This mode is also used by the I/O instructions to specify a "port" and always indicates a 16-bit I/O address held in a single word register.

The Indirect Register mode may save space and reduce run time when consecutive locations are referenced. This mode can also be used to simulate more complex addressing modes, since addresses can be computed before the data is accessed.

Direct Addressing

In Direct Address mode, the data processed is found at the address specified as an operand.

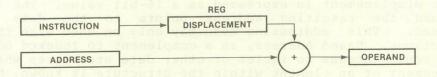


THE OPERAND VALUE IS THE CONTENTS OF THE LOCATION WHOSE ADDRESS IS IN THE INSTRUCTION

This mode is also used by the I/O instructions to specify a "port" and always indicates a 16-bit I/O address. This mode is also used by Jump and Call instructions to specify the address of the next instruction to be executed (actually, the address serves as an imediate value that is loaded into the program counter). The address specified is limited to one word in Nonsegmented mode and in Segmented mode using the short offset (Figure 1-4). Segmented addresses using a long offset require a long word.

Indexed Addressing with avidated about assubba to avid

In Indexed Address mode, the instruction processes data located at an indexed address in memory. The indexed address is computed by adding the address specified in the instruction to a "displacement" or "index". The index is contained in a word register, also specified in the instruction. Indexed Addressing allows random access to tables or other complex data structures where the address of the base of the table is known, but the particular element index must be computed by the program.

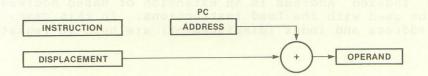


THE OPERAND VALUE IS THE CONTENTS OF THE LOCATION WHOSE ADDRESS IS THE ADDRESS IN THE INSTRUCTION, OFFSET BY THE CONTENTS OF THE REGISTER

Any word register can be used as the index register $except\ R0$. The address operand is limited to one word in Nonsegmented mode, but can be one or two words in Segmented mode, depending on whether a long or short offset is used in the address (Figure 1-4).

Relative Addressing

In Relative Address mode, the data processed is found at an address relative to the current instruction. The instruction specifies a two's complement displacement which is added to the program counter to form the target address. The program counter setting used is the address of the first instruction following the current relative instruction.

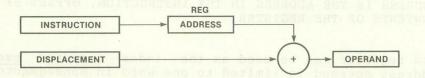


THE OPERAND VALUE IS THE CONTENTS OF THE LOCATION WHOSE ADDRESS IS THE CONTENTS OF PC OFFSET BY THE DISPLACEMENT IN THE INSTRUCTION

As with Direct Address mode, Relative Address is also used by certain program control instructions to specify the address of the next instruction to be executed (specifically, the result of the addition of the program counter and the displacement is loaded into the program counter).

Based Addressing

Based Address mode is similar to Indexed Address mode. In Based Address, however, the register operand specifies the base address and the displacement is expressed as a 16-bit value. The two are added and the resulting address points to the data to be processed. This addressing mode may only be used with the Load instructions. Based Address, as a complement to Indexed Address, allows random access to tables or other data structures where the displacement of an element within the structure is known, but the base of the particular structure must be computed by the program.

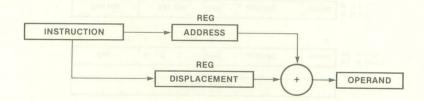


THE OPERAND VALUE IS THE CONTENTS OF THE LOCATION WHOSE ADDRESS IS THE ADDRESS IN THE REGISTER, OFFSET BY THE DISPLACEMENT IN THE INSTRUCTION.

Any word register (or register pair in Segmented mode) can be used for the base address except RO or RRO. In Segmented mode, Based Address allows access to locations whose segment numbers are not known at assembly time.

Based Indexed Addressing

Based Indexed Address is an extension of Based Address, and may only be used with the Load instructions. In this case, both the base address and index (displacement) are held in registers.



THE OPERAND VALUE IS THE CONTENTS OF THE LOCATION WHOSE ADDRESS IS THE ADDRESS IN THE REGISTER, OFFSET BY THE DISPLACEMENT IN THE REGISTER.

Any word register (or register pair in segmented mode) can be used as the base address except $\underline{R0}$ or $\underline{RR0}$. Any word register can be used as the index except $\underline{R0}$ or $\underline{RR0}$.

1.5.5 Instruction Formats

Z8000 instructions may occupy one to four words, depending on the number of operands and the number of words needed to form operand addresses or immediate values. Figure 1-10 shows some of the formats for typical instructions and the main fields within these formats.

Regardless of an instruction's length, its first byte contains the operation code (also referred to as the "opcode"). In addition to specifying the operation, this field also indicates the addressing mode (bits 14-15), and the word or byte data type (bit 8), as applicable. Instructions can designate zero or more operands explicitly. If the operands designate general-purpose registers, the register address(es) are usually specified in the second byte of the instruction. The format of these fields are detailed in Section 3.

1.6 PROGRAM CONTROLS

In addition to the general-purpose registers described in Section 1.5.2, the Z8000 CPU has several control registers containing status flags, control bits, the Program Counter, a pointer to the Program Status Area, and a memory refresh register.

The status flags, control bits, and program counter are referred to collectively as the <u>Program Status</u> (PS). The Program Status is contained in two or four words, depending on whether the nonsegmented or segmented version of the Z8000 is used (Figure 1-11). The status flags and control bits are referred to collectively as the <u>Flags and Control Word</u> (FCW).

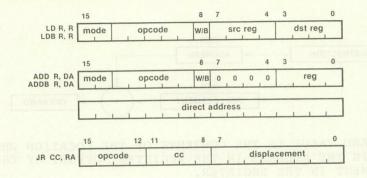


Figure 1-10 Typical Instruction Formats (Nonsegmented)

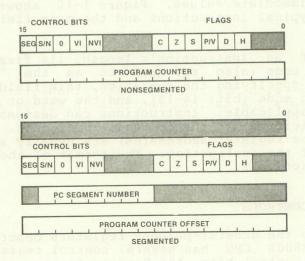


Figure 1-11 Program Status Blocks

1.6.1 Program Counter

The Program Counter (PC) contains the address in program memory of the next instruction to be executed. As shown in Figure 1-11, the Program Counter is 16 bits for the nonsegmented version and 32 bits for the segmented version. The segmented address format is described in Section 1.4.1.

1.6.2 Status Flags

Status Flags (FLAGS) can be used to determine the outcome of certain operations and to redirect the flow of the program as necessary. The program status has six flags for the use of the programmer and the Z8000 processor:

- Carry (C) Zero (Z)
- Sign (S)
- Parity/Overflow (P/V)
- Decimal Adjust (D)
- Half Carry (H)

Z8000 CPU control instructions allow the programmer to set, reset (clear), or complement any or all of the first four flags. The half-carry and decimal-adjust flags are used only by the Z8000 for BCD arithmetic corrections.

The FLAGS register can be separately loaded by the Load Control Register (LDCTLB) instruction without disturbing the control bits in the other byte of the FCW. The C, Z, S, and P/V flags can also be used with branching instructions to provide up to 21 conditional tests and as loop controls in string instructions.

The carry (C) flag, when set, generally indicates a carry out of the high-order bit position of a register being used as an accumulator. For example, adding two 8-bit numbers causes a carry out of bit 7 and sets the carry flag:

> A faeupeadue ed Bit da da 7 6 5 4 3 2 1 0 225 1 1 1 0 0 0 0 1 64 0 1 0 0 0 0 0 0 289 0 0 1 0 0 0 0 1 1 = carry flag

The $\underline{\text{zero}}$ (Z) flag is set when the result register's contents are zero following certain operations.

The <u>sign</u> (S) flag is set to one when the most significant bit of a <u>result</u> register contains a one (a negative number in two's-complement notation) following certain operations.

The <u>overflow</u> (V) flag, when set, indicates that a two's-complement number in a result register has exceeded the largest or is less than the smallest number that can be represented in a two's-complement notation. This flag is set as the result of an arithmetic operation. Consider the following example:

The result in this case (-95 in two's complement notation) is incorrect, thus the overflow flag would be set.

The same bit acts as a <u>parity</u> (P) flag following logical instructions for byte operands only. The number of one bits in the register is counted and the flag is set if the total is even (that is, P=1). If the total is odd, the flag is reset (P=0).

The <u>decimal-adjust</u> (D) flag is used for BCD arithmetic. Since the algorithm for correcting BCD operations is different for addition and subtraction, this flag is used to specify what kind of instruction was executed so that the subsequent Decimal Adjust (DAB) instruction can perform its function correctly.

The half-carry (H) flag indicates a carry out of, or a borrow into bit 3 as the result of adding or subtracting two BCD digits. This flag is used by the DAB instruction to convert the binary result of a previous decimal addition or subtraction into the correct decimal (BCD) result.

Neither the decimal-adjust nor the half-carry flag is normally accessed by the programmer. The specific operations affecting the flags are detailed in Section 3 and listed in Appendix A.

1.6.3 Control Bits

The <u>control bits</u> are used to enable various interrupts or operating modes. The nonsegmented Z8000 uses three control bits; the segmented version has four (Figure 1-11). These bits are:

• Vectored Interrupt Enable (VI)

Non-Vectored Interrupt Enable (NVI)

• Segmentation Mode (SEG), used only by the segmented Z8000

System/Normal Mode (S/N)

The two interrupt control bits (VI and NVI) are set and reset by the Enable Interrupt (EI) and Disable Interrupt (DI) instructions. When the control bit is set to one, the appropriate interrupt is enabled; otherwise, it is disabled. Interrupts and interrupt handling are described further in the following two sections.

The Segmentation mode bit has meaning only for the segmented version of the Z8000. The setting of this bit indicates whether the Z8000 is running in Segmented (=1) or Nonsegmented mode (=0). The bit is set and reset by the LDPS or LDCTL instructions.

The System/Normal bit indicates the operating mode of the program. When set to one, System mode is in effect; when reset to zero, Normal mode is in effect. This bit can be explicitly set or reset by the LDPS or LDCTL instructions, or implicitly changed by the occurrence of an interrupt or trap (Section 1.6.4) when a new program status is loaded. The programmer can also enter System mode from Normal mode (and change the setting of this control bit) by issuing a System Call (SC) instruction.

Any control bit can be changed by the occurrence of an interrupt or trap, and then restored to its previous setting by terminating the interrupt handler procedure with an Interrupt Return instruction (IRET). The Interrupt Return pops the saved program status off the system stack.

1.6.4 Interrupts and Traps

<u>Interrupts</u> are asynchronous events and are typically triggered by peripheral devices needing attention. The three kinds of interrupts are:

- Non-Maskable interrupt (NMI)
- Vectored interrupt (VI)
- Non-Vectored interrupt (NVI)

Non-Maskable interrupts cannot be disabled, and are usually reserved for critical external events that require immediate attention. Vectored interrupts cause a 16-bit value output by the interrupting device to be read from the data bus (ADO-AD15). This "vector" value is used to select a particular interrupt procedure to automatically branch to (Section 1.6.5). Non-Vectored interrupts are all handled by the same interrupt procedure, which may "poll" the external devices to determine which one requires attention.

Traps, on the other hand, are synchronous events and usually indicate some special programming error or condition. They are triggered by specific instructions and recur each time the instruction is executed with the same set of data. The four kinds of traps are:

• Unimplemented instructions

Privileged instructions in Normal mode

Segmentation violations

• System call

During the execution of an instruction, one of three kinds of error conditions can arise. An "illegal instruction exception" signifies that that the binary code of the current instruction is an illegal value for the architecture of the Z8000. The result of this error is undefined, so the programmer must not use binary instruction values other than those defined in the instruction set of Section 3.

An "unimplemented instruction exception" signifies that the binary code of the current instruction is defined by the Z8000 architecture, but is not currently implemented by the hardware (Section 3.4). In this case, an Unimplemented Instruction trap occurs, which allows system software to either simulate the execution of the instruction or abort the program.

An "operation exception" signifies that the binary code of the current instruction is valid, but the operand specification or execution of the instruction is architecturally invalid. For example, an attempt to execute a privileged instruction in Normal mode will cause a trap. A segmentation violation, such as using an offset larger than the defined length of the segment, will cause the MMU to signal a Segmentation trap. (Segmentation traps occur only with the segmented Z8000.) All other operation exceptions, such as specifying an odd address for a word data value, are considered programmer errors and the results are undefined.

The System Call instruction (SC) provides a controlled access from Normal mode software to System mode software, and is handled in a manner similar to the other traps.

Because it is possible for several interrupts or traps to occur simultaneously, a priority for selecting which kind of interrupt or trap is honored first is established by the Z8000 architecture. The descending priority order of the traps and interrupts is: internal trap (unimplemented instructions, privileged instructions and the System Call instruction), Non-Maskable interrupt, Segmentation trap, Vectored interrupt, and Non-Vectored interrupt.

Interrupts and traps are handled similarly by the Z8000 hardware. At the start of the interrupt or trap sequence, the Z8000 is forced into System mode. In addition, the segmented version of the Z8000 is forced into Segmentation mode, regardless of the current mode. The program status information in effect just prior to the interrupt or trap is pushed onto the system stack. An additional word, indicating the "identifier" for the interrupt or trap, is also pushed onto the system stack, where it can be accessed by the interrupt or trap handler (Figure 1-12). The new program status is set up as described in the following section (which includes loading the program counter with the starting address of the procedure for servicing the interrupt or trap), and then control is transferred to the service procedure.

For internal traps, the "identifier" stored in the system stack is the first word of the instruction causing the trap. For interrupts and external traps (such as a segmentation trap), the "identifier" stored is the 16-bit value read from the data bus (ADO-AD15) at the start of the interrupt acknowledge sequence. In the case of Vectored interrupts, this value is also used to select the appropriate service procedure, as explained in the following section.

1.6.5 Program Status Area

As part of a system software configuration, the user must provide service procedures to handle the various interrupts and traps. The procedures are accessed through their respective program status blocks (Figure 1-11) which determine the new program status set up when program execution is interrupted. These status words reside in a reserved area of memory called the Program Status Area, also established by the programmer, and should be arranged as shown in Figure 1-13. The ordering is important, because the specific program status block selected (and consequently the service procedure selected) is determined implicitly from the kind of interrupt or trap that occurred. Note that the size of each program status block depends on the version of the Z8000 (two words for the nonsegmented and four words for the segmented version).

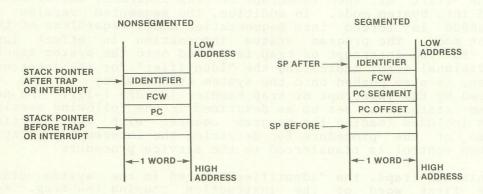


Figure 1-12 Format of Saved Program Status in the System Stack

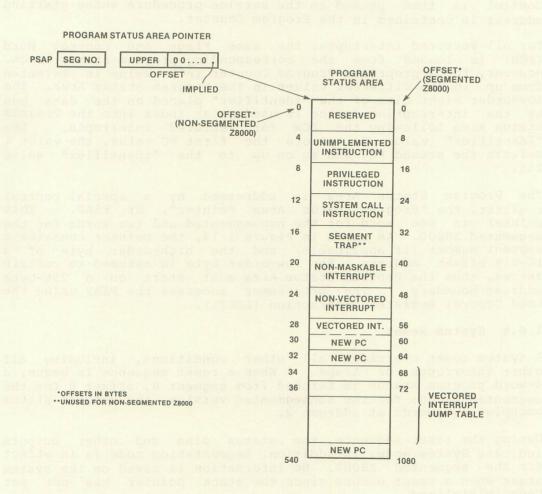


Figure 1-13 Program Status Area

For each kind of interrupt or trap other than a Vectored interrupt, there is a single program status block that is automatically loaded into the Program Status registers (which includes the Flags and Control Word and the Program Counter). Control is then passed to the service procedure whose starting address is contained in the Program Counter.

For all Vectored interrupts, the same Flags and Control Word (FCW) is loaded from the corresponding program status block. However, the appropriate Program Counter (PC) value is selected from up to 256 different values in the Program Status Area. The low-order eight bits of the "identifier" placed on the data bus by the interrupting device is used as an index into the Program Status Area following the FCW for Vectored interrupts. The "identifier" value 0 selects the first PC value, the value 1 selects the second PC, and so on up to the "identifier" value 255.

The Program Status Area is addressed by a special control register, the "Program Status Area Pointer", or PSAP. This pointer is one word for the nonsegmented and two words for the segmented Z8000. As shown in Figure 1-14, the pointer contains a segment number (if applicable) and the high-order byte of a 16-bit offset address. The low-order byte is assumed to contain zeroes, thus the Program Status Area must start on a 256-byte address boundary. The programmer accesses the PSAP using the Load Control Register instruction (LDCTL).

1.6.6 System Reset

A system reset overrides all other conditions, including all other interrupts or traps. When a reset sequence is begun, a 4-word program status is fetched from segment 0, offset 0 for the segmented Z8000; for the nonsegmented version, the program status occupies two words at address 2.

During the reset sequence, the status pins and other outputs indicate System mode; in addition, Segmentation mode is in effect for the segmented Z8000. No information is saved on the system stack when a reset occurs since the stack pointer has not yet been initialized.

1.6.7 Memory Refresh

The refresh control register (REFRESH) is a 16-bit counter used to refresh dynamic memory automatically. A special refresh memory access is made at programmable intervals and is otherwise invisible to the currently executing program. This register has the format shown in Figure 1-15 and, like the other control registers, can be programmed using the Load Control Register (LDCTL) instruction.

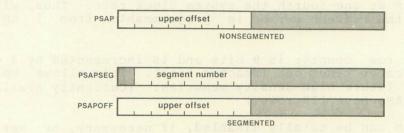


Figure 1-14 Program Status Area Pointer

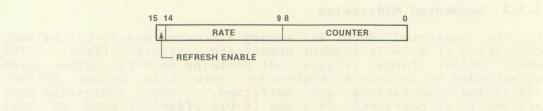


Figure 1-15 Refresh Register

The refresh rate (the time between successive refreshes) is determined by the 6-bit prescaler. This is a modulo-n counter (n = 1-64) driven at one-fourth the system clock rate. Thus, with a 4 MHz clock, the refresh period is programmable from 1 to 64 microseconds.

The refresh row counter is 9 bits and is incremented by 2 each time the prescaler times out (reaches zero). This allows up to 256 rows for future high-density memories. (Currently available 16K dynamic RAMs have 128 rows.)

Memory refresh can be totally disabled, if necessary, by setting bit 15 of the refresh register to zero.

1.7 ADDRESS ARITHMETIC

1.7.1 Nonsegmented Addressing

In the nonsegmented Z8000, all addresses are 16-bit values. When addressing mode arithmetic is performed, the result is always taken modulo 65536. No carry is generated or any other indication that the result of addition or other addressing operations may have overflowed 16 bits. Thus in Indexed, Relative, Based and Based Indexed addressing modes, or in autoincrement and autodecrement instructions, the resulting address always remains within the 0 to 65535 addressing space.

1.7.2 Segmented Addressing

In the segmented Z8000, all memory addresses are 23-bit values, consisting of a 7-bit segment number and a 16-bit offset. The short offset format (Figure 1-4) contains an 8-bit offset which is extended to include 8 high-order zero bits before further addressing computations are performed. When addressing mode arithmetic is performed, only the 16-bit offset is used so that the result is taken modulo 65536 as in nonsegmented addressing. No carry out of the 16-bit offset is generated, nor is any other indication that the result of addition or other addressing operations may have overflowed 16 bits. In other words, the segment number is not affected by addressing arithmetic. Thus in Indexed, Relative, Based and Based Indexed addressing modes, or in autoincrement or autodecrement instructions, the resulting address always remains within the same segment with the offset in the range 0 to 65535.

Nonsegmented programs can be run in one segment of the segmented Z8000 by setting the Segmentation mode control bit to zero with the LDPS or LDCTL instructions. In this case, the current value of the Program Counter's segment number is used for all memory accesses. While "running the segmented Z8000 in Nonsegmented mode", the rules of nonsegmented addressing mode arithmetic apply.

All I/O addresses are 16-bit values, regardless of whether memory addresses are segmented or nonsegmented. Therefore, all I/O address arithmetic follows the rule for nonsegmented addressing.

Section 2 **Z8000 Assembler Conventions**

2.1 ASSEMBLER OVERVIEW

The Z8000 microprocessor is programmed in a symbolic assembly language (PLZ/ASM). This marks a significant improvement over coding in binary notation. The operation codes for assembly-language statements are easily memorized (for example, SUB for Subtract and LD for Load). In addition, meaningful symbolic names can be assigned to program addresses and data (for example, ALLOCATE as the label of the first statement in a storage allocation procedure).

A Z8000 source module is made up, for the most part, of such assembly language statements. These statements are then translated by the Z8000 assembler into an object module that can either be separately executed by the Z8000 microprocessor, or can be linked with other object modules to form a complete program. Because the assembler has some high-level features, a source module can also include PLZ constructs such as DO and IF statements. The user can also embed assembler directives, which control the operation of the assembler, in the source module. High-level statements and assembler directives are discussed in Sections 4 and 5.

Depending on the assembler directives used, addresses within an object module or program can be absolute (meaning addresses in the source program correspond exactly to Z8000 logical memory addresses) or relocatable (meaning addresses can be assigned relative to some logical base address at a later time). Object modules should be made relocatable wherever possible. This facilitates both the ability to link with other object modules as well as the ability to load object programs anywhere in memory. It also allows the creation of libraries of commonly used procedures (including math or input/output routines) that can be linked selectively into several programs as desired.

Operation of the assembler, module linkage, address relocation, and program execution are the subject of the Z8000 Assembler User's Guide.

2.2 ASSEMBLY LANGUAGE STATEMENT FORMAT

The most fundamental component of a PLZ/ASM program is the assembly language statement consisting of an instruction and its operand(s). The instruction describes an action to be taken; the operand(s) supplies the data to be acted upon.

An assembly language statement can include four fields:

- Statement label(s)
- An instruction
- Operand(s)
- Comments

The statement label and comment fields are always optional. The statement has zero or more operands, depending on the instruction selected. The following statements have the same effect in a Z8000 program, but the second is more descriptive (and consequently more helpful in program debugging).

| Label | Instruction | Operand(s) | Comment | |
|------------|-------------|----------------------------|---------------------------------|--|
| INITCOUNT: | LD LD | COUNT, #255 COUNT, #255 | !Load COUNT with initial value! | |

Each of the elements of a PLZ/ASM program must be separated from other elements by one or more delimiters. A delimiter is one of the characters: space (blank), comma, semicolon, tab, carriage return, line feed, or form feed. Note that carriage return is treated just like any other delimiter, so that a single statement may span several lines, or several statements may appear on a single line. The delimiter used in a specific situation is up to the programmer. For the sake of illustration, this manual uses blanks to separate statement fields and commas to separate operands.

2.2.1 Program Labels and Identifiers

Any assembly language (or high-level) statement in a 28000 program can be preceded by any number of labels. Any statement referenced by another statement must be labeled. A label consists of an identifier followed by a colon (:) in the form:

labell: label2: ... labeln: statement

A PLZ/ASM identifier can contain up to 127 characters, of which the first must be a letter. The remaining characters can be letters, digits, or the special character underscore (_). Letters can be capitalized or lower-cased, but each time an identifier is used, it must be written in exactly the same way. The following are valid identifiers:

START_UP_ROUTINE
Program_Initialization
A
Loop_12
N1
sort

In addition to their statement-labeling function, identifiers also serve as symbolic names for constants (Section 2.3.2), data variables (Section 2.3.3), and procedures (Section 5.2.2). Certain identifiers serve as PLZ/ASM keywords and should not be used as programmer-defined identifiers (see Appendix D).

An identifier can be associated with only one item within the scope of its definition. Section 4.2.5 explains the scope of identifiers, including the scope of labels. Labels are accessible within the module in which they are defined, and are not accessible outside that module unless specifically declared to be GLOBAL or EXTERNAL.

2.2.2 Instruction

The instruction is the assembly-language mnemonic describing a specific action to be taken. The instruction must be separated from its operand(s) by a delimiter.

LD R5, R10 !Load register 5 from register 10!

CLR R10 !Clear register 10!

Many of the operations of the Z8000 can be applied to word, byte, and long operands. A simple naming convention has been adopted to distinguish the size of the operands for these particular instructions: the suffix "B" designates a byte instruction, the suffix "L" designates a long word instruction, and no suffix designates a word instruction:

ADD RO, R1 !Add word operands!

ADDB RHO, RLO !Add byte operands!

ADDL RRO, RR2 !Add long operands!

2.2.3 Operand Field

Depending on the instruction specified, this field can have zero or more operands. If two or more operands are needed, each must be separated by a delimiter.

IRET !No operand!

COM R10 !One operand!

ADD R6, #210 !Two operands!

LDM R2, SAVEREG, #5 !Three operands!

CPD R2, @R6, R1, Z !Four operands!

Operands supply the information the instruction needs to carry out its action. An operand can be:

- Data to be processed (immediate data);
 - The address of a location from which data is to be taken (source address);
 - The address of a location where data is to be put (destination address);
 - The address of a program location to which program control is to be passed;
 - A condition code, used to direct the flow of program control.

Although there are a number of valid combinations of operands, there is one basic convention to remember: the destination operand always precedes the source operand. Refer to the specific instructions in Section 3 for valid operand combinations.

Immediate data can be in the form of a $\underline{\text{constant}}$, an $\underline{\text{address}}$, or an $\underline{\text{expression}}$ (constants and/or addresses combined by $\underline{\text{operators}}$). Each of these forms is described in Section 2.3.

| LD | RO, | #K | !Load constant K into reg 0! |
|-----|-----|------------|--|
| LD | RO, | #COUNTER | !Load address of COUNTER into reg 0! |
| ADD | R0, | #CON/3 + 5 | !Add value of expression (CON/3 + 5) to contents of reg 0! |

Source, destination, and program addresses can also take several forms. PLZ/ASM addressing modes are described in Section 2.4. Some examples are:

| LD | RO, @R5 squ shu | !Load word value whose address is in register 5 into register 0! |
|-----|-----------------|---|
| LDB | RH5, VAR1 | !Load byte value located at address labeled VARl into register RH5! |
| LDL | RR10, VAR1 + 1 | !Load long value at location following that addressed by VAR1 into register pair 10-11! |
| JP | Z, LOOP1 | !Jump to program address labeled LOOP1 if zero flag (Z) is set! |
| JP | NZ, LOOP1 + 6 | !Otherwise, jump to location six bytes after LOOP1! |

Condition codes are listed in Section 3.2.1.

2.2.4 Comments

Comments are used to document program code as a guide to program logic and also to simplify present or future program debugging. Comments can be inserted anywhere a program delimiter may appear. Comments are bounded by exclamation points (!) and can contain any characters except the exclamation point itself.

!Module 3, Changed 7-25-78!

RES R15, #3 !?!

A single comment can cross line boundaries; that is, carriage returns can occur within a comment.

2.3 ARITHMETIC OPERANDS

2.3.1 Run-Time vs. Assembly-Time Arithmetic

Arithmetic is performed in two ways in an assembly language program. Run-time arithmetic is done while the program is actually executing.

SUB R10, R12 !Subtract the contents of register 12 from the contents of register 10!

Assembly-time arithmetic is done by the assembler when the program is assembled and involves the evaluation of arithmetic expressions in operands, such as the following:

LD R0, #(22/7 + X)

JP Z, LOOP1 + 12

ADD R2, HOLDREG-1

Assembly-time arithmetic is more limited than run-time arithmetic in such areas as signed vs. unsigned arithmetic and the range of values permitted.

Only unsigned arithmetic is allowed in assembly-time expression evaluation. Run-time arithmetic uses both signed and unsigned modes, as determined from the assembly-language instruction specified and the meaning attached to operands by the programmer.

All assembly-time arithmetic is computed using 32-bit arithmetic, "modulo 4,294,967,296" (2 raised to the thirty-second power). Values greater than or equal to 4,294,967,296 are divided by 4,294,967,296 and the remainder of the division is used as the result. Depending on the number of bits required by the particular instruction, only the rightmost 4, 8, 16, or 32 bits of the resulting 32-bit value are used. If the result of assembly-time arithmetic is to be stored in four bits, the value is taken "modulo 16" to give a result in the range 0 to 15. If the result is to be stored in a single byte location, the value is taken "modulo 256" to give a result in the range 0 to 255 (or -128 to 127 if signed representation is intended). If the result is to be stored in a word, the value is taken "modulo 65536" to give a result in the range 0 to 32767 if signed representation is intended).

LDB RL4, #X+22

!Result of (X+22) must be in range -128 to 255!

JP X+22

!Modulo 65536. Result is the address 22 bytes beyond X!

ADDL RR12, #32000*MAX !Result of (32000*MAX) is taken modulo 4,294,967,296!

2.3.2 Constants

A constant value is one that doesn't change throughout the program. Constants may be expressed as numbers, as character sequences, or as a symbolic name representing a constant value.

Numbers can be written in decimal, hexadecimal, binary, or octal notation. The latter three are preceded by a percent sign (%) and, in the case of binary and octal, by a base specifier enclosed in parentheses. If a number has no prefix, decimal is assumed.

decimal 10 hexadecimal 810 %AFOF hexadecimal hexadecimal 8(8)70 octal

A character sequence is a sequence of one or more characters enclosed in single quote marks. Any ASCII character (except a percent sign or single quote) can be included in the character sequence.

'A'
'This is a character sequence'

A character can also be represented in a character sequence in the form "%hh," where "hh" is the hexadecimal equivalent of the ASCII code for the character. (See Appendix E for the ASCII character set and its hexadecimal equivalents.)

'Here is an ESC character: %1B'

For convenience, certain ASCII characters have been assigned shorter, more mnemonic codes as follows:

%L or %l Linefeed

signed representation is intended). %T or %t Tab Carriage Return %R or %r Page (Form Feed) %P or %p Percent Sign 88 %Q or %q Single Quote

Example:

'First line%rSecond line%r'
'Quote%Qinside a quote%Q'

A constant can be assigned a symbolic name by a constant definition (CONSTANT) statement. A symbolic identifier, once associated with a constant value, retains that value through the entire program module.

Constant symbols are defined by the CONSTANT statement in the form shown below. Identifiers follow the rules outlined in Section 2.2.1. The special character pair ":=" can be read "is defined as".

CONSTANT

CONSTANT

REC_LENGTH := 64

BUFFER_LENGTH := 4*RECLENGTH

SEMICOLON := ';' SEMICOLON := 1000000 BIGNUMBER smallnumber := -1

It is also possible to create a new symbol which will be treated the same as any reserved keyword (see Appendix D for a list of reserved keywords). If the symbol on the right side of the ':=' in a CONSTANT statement is a keyword, then the symbol on the left side can be used thereafter any place the keyword would be valid. One important use of this ability is to "rename" a register such as R5 with a more meaningful name such as SUBTOTAL (of course, the symbol R5 can still be used). The programmer is cautioned, however, that renaming keywords in general can lead to confusing and difficult to maintain programs.

2.3.3 Data Variables

A data variable can be thought of as a "container" that can hold different values from time to time. Just as a physical 8-ounce container can hold 0-8 ounces of liquid, an 8-bit (BYTE or SHORT INTEGER) variable can hold values in the range 0 to 255 if unsigned, or -128 to 127 if signed two's complement representation is intended. A 16-bit (WORD or INTEGER) variable can hold values in the range 0 to 65535 if unsigned, or -32768 to 32767 if signed. Similarly, a 32-bit (LONG or LONG INTEGER) variable can hold values in the range 0 to 4,296,967,296 (or -2,148,483,648 to 2,148,483,647 if signed two's complement representation is intended).

While it is suggested that BYTE, WORD, and LONG variables be used for unsigned values, and SHORT INTEGER, INTEGER, and LONG INTEGER variables be used for signed values, there are no restrictions on whether a particular variable is signed or unsigned. In other words, BYTE and SHORT INTEGER are treated as equivalent, as are WORD and INTEGER, as are LONG and LONG INTEGER, with the appropriate interpretation left entirely to the programmer.

A data variable name can be associated with a data memory location; the value of the variable is the contents of that location at the time the variable is referenced. A data variable is a symbolic identifier and follows the rules for identifiers in Section 2.2.1.

> T.D R5, MPLIER

!Load the value contained in the location symbolized by MPLIER!

ADD R5, 4 + SUBTOTAL

!Add the value contained in the location 4 bytes after the location addressed by SUBTOTAL to the contents of register 5!

If a data variable operand is preceded by "#," it is treated as immediate data and the value used is the data address associated with the variable, $\underline{\text{not}}$ the $\underline{\text{contents}}$ of the location. For example, suppose location 50 has the symbolic name COUNTER and contains the bit pattern 11111111 (decimal 255).

> LDB RLO, COUNTER

!255 is loaded into RLO!

LDB

RLO, #COUNTER !50 is loaded into RLO!

LDB

RLO, COUNTER - 5 !Contents of location 45 are

loaded into RLO!

LDB RLO, #COUNTER - 5 !45 is loaded into RLO!

Every data variable name has a type and scope associated with it, as well as a value. The type and scope (and, optionally, the initial value) are defined in a variable declaration statement like the following:

> INTERNAL SWITCH1 BYTE

In this example, "INTERNAL" is the scope of the variable SWITCH1, and "BYTE" is its type.

Variables can have GLOBAL, EXTERNAL, INTERNAL, or LOCAL scope. They can be one of the "simple" types BYTE or SHORT_INTEGER (for 8-bit values), WORD or INTEGER (for 16-bit values), or LONG or LONG_INTEGER (for 32-bit values). They can also be one of the "structured" types ARRAY or RECORD. (Section 5.3.4 discusses variable declaration in more detail.)

2.3.4 Expressions and Operators

Expressions are formed using arithmetic, logical, shift, and relational operators in combination with constants and variables. These operators allow both unary (single-operand) and binary (two-operand) expressions, as shown below.

Arithmetic Operators. The arithmetic operators are as follows:

| Operator | Operation |
|-----------------|---------------------------------|
| seen from ti+ | onary pracy britary addressin |
| for darking and | Unary minus, binary subtraction |
| * | Unsigned multiplication |
| / | Unsigned division |
| MOD | Unsigned modulus |

The division operator (/) truncates any remainder. The MOD operator returns the remainder from dividing its operands.

17/4 = 4

17 MOD 4 = 1

If zero is specified as the right operand for either of these division operators, the result is undefined.

Examples:

ADD R5, #-3 !A minus 3 is added to register 5!

ADD R5, #K + (5*3) !Value of constant K + 15 is added to register 5!

Note that expressions containing these operators are evaluated at assembly time and, consequently, the arithmetic performed is unsigned. Signed arithmetic can still be done at run time.

Logical Operators. The logical operators are as follows:

Operation Operation

LNOT (Unary) Logical complement

LAND Logical AND

LOR Logical OR

LXOR Logical EXCLUSIVE OR

LNOT simply complements the bit pattern of its (single) operand. All one bits are changed to zero and vice-versa.

LD R2, #LNOT MASK !Reverse the bits in a mask and load into reg 2!

The effect of LAND, LOR, and LXOR can be seen from the following examples. Although 32-bit arithmetic would actually be done by the assembler, 8-bit arithmetic is shown for clarity. Assume two constants A and B have the bit patterns 11110000 and 01010101, respectively. The expressions:

A LAND B A LOR B A LXOR B

will result in the following evaluations of the operands:

| LAND | 11110000 01010101 | LOR | 11110000 LXOR 01010101 | | 11110000 |
|------|----------------------|-----|---------------------------|--|----------|
| | 01010000 | | 11110101 | | 10100101 |

LAND sets a one bit whenever both ANDed bits are one; LOR sets a one bit whenever either ORed bit is one; LXOR sets a one bit when the two EXCLUSIVE-ORed bits are different.

The assembly-time logical operations performed by LNOT, LAND, LOR, and LXOR can also be done at run time by the Z8000 instructions COM, AND, OR, and XOR. The assembly-time operations require less code and register manipulation. The run-time operations allow greater flexibility, however. For example, they can operate on registers (variables) whose contents are not known at assembly time, as well as on known constant values.

Shift Operators. The shift operators are as follows:

SHR Logical shift right SHL Logical shift left

When used in expressions, the shift operators have the form

d operator n

where "d" is the data to be shifted and "n" specifies the number of bits to be shifted. Vacated bits are replaced with zeros. For example, if the constant PRODUCT is equal to 10110011, the statement

> LDB RLO, #(PRODUCT SHL 2)

would load the value 11001100 into register RLO.

If the second operand supplied is negative (that is, if the sign bit is set), it has the effect of reversing the direction of the shift.

ADD PRODUCT, #(MPLIER SHR -1) !MPLIER is shifted left one bit position!

Relational Operators. The relational operators are as follows:

Less than

by the pr=>ramer Less than or equal

Equal Miled by O. F. H. H. Not equal ()

>= Greater than or equal > Greater than

These six relational operators return a logical TRUE value (all ones) if the comparison of the two operands is true, and return a logical FALSE value (all zeros) otherwise. The operators assume both operands are unsigned.

LD R0, #(1=2)!Reg 0 is loaded with zeros! LD R0, #(2+2) < 5 !Reg 0 is loaded with ones!

<u>Precedence of Operators</u>. Expressions are generally evaluated left to right with operators having the highest precedence evaluated first. If two operators have equal precedence, the leftmost is evaluated first.

The following lists the PLZ/ASM operators in order of precedence:

- Unary operators: +, -, LNOT
- Multiplication/Division/Shift/AND: *, /, MOD, SHR, SHL, LAND
- Addition/Subtraction/OR/XOR: +, -, LOR, LXOR
- Relational operators: <, <=, =, <>, >=, >

Parentheses can be used to change the normal order of precedence. Items enclosed in parentheses are evaluated first. If parentheses are nested, the innermost are evaluated first.

$$20/5 - 12/3 = 0$$

$$20/(5 - 12/3) = 20$$

Modes of Arithmetic Expressions. All arithmetic expressions have a mode associated with them: absolute, relocatable, or external. These modes are defined in detail in Section 4, following the explanation of the concepts of "scope" and "program relocatability".

2.3.5 Segmented Address Operators

Two special operators are provided to ease the manipulation of segmented addresses. While addresses can be treated as a single value with a symbolic name assigned by the programmer, occasionally it is useful to determine the segment number or offset associated with a symbolic name.

The "SEG" unary operator is applied to an address expression which contains a symbolic name associated with an address, and returns a 16-bit value. This value is the 7-bit segment number associated with the expression and a one bit in the most significant bit of the high-order byte, and all zero bits in the low-order byte (Section 1.4.1).

The "OFFSET" unary operator is applied to an address expression and returns a 16-bit value which is the offset value associated with the expression.

Example: LD R2, #SEG PTR LD R3, #OFFSET PTR

!Load segmented address of PTR into register pair RR2, which is functionally equivalent to the following statement!

LDL RR2, #PTR

Because of the special properties of these address operators, no other operators may be applied to a subexpression containing a SEG or OFFSET operator, although other operators can be used within the subexpression SEG or OFFSET are applied to:

SEG (PTR+4) {Valid} (SEG PTR)+4 {Invalid} -(OFFSET PTR) {Invalid}

2.4 Z8000 ADDRESSING MODES

With the exception of immediate data and condition codes, all assembly-language operands are expressed as addresses: register, memory, and I/O addresses. The various address modes recognized by the Z8000 assembler are as follows:

- Immediate Data
- Register
- Indirect Register
- Direct Address
- Indexed Address
- Relative Address
- Based Address
- Based Indexed Address

Special characters are used in operands to identify certain of these address modes. The characters are:

- "R" preceding a word register number;
- "RH" or "RL" preceding a byte register number;
- "RR" preceding a register pair number;
- "RQ" preceding a register quadruple number;
- "@" preceding an indirect-register reference;
- "#" preceding immediate data;
- "()" used to enclose the displacement part of an indexed, based, or based indexed address;
- "\$" signifying the current program counter location, usually used in relative addressing.

The use of these characters is shown in the following sections.

Not every address mode can be used by every instruction. The individual instruction descriptions in Section 3 tell which address modes can be used for each instruction.

2.4.1 Immediate Data

The operand value used by the instruction in Immediate Data addressing mode is the value supplied in the operand field itself.

Immediate data is preceded by the special character "#" and may be a constant (including character constants and symbols representing constants) or an expression as described in Section 2.3.4. Immediate data expressions are evaluated using 32-bit arithmetic. Depending on the instruction being used, the value represented by the rightmost 4, 8, 16, or 32 bits is actually used. An error message is generated for values that overflow the valid range for the instruction.

LDB RHO, #100 !Load 100 into byte register RHO!

LDL RRO, #%8000 * REP COUNT

!Load the value resulting from the multiplication of %8000 and the value of constant REP COUNT into register pair RRO!

If a variable name or address expression is prefixed by "#", the value used is the address represented by the variable or the result of the expression evaluation, not the contents of the corresponding data location. In Nonsegmented mode, all address expressions result in a 16-bit value.

For segmented addresses, the assembler automatically creates the proper format for a long offset address which includes the segment number and the offset in a 32-bit value (Section 1.4.1). It is recommended that symbolic names be used wherever possible, since the corresponding segment number and offset for the symbolic name will be automatically managed by the assembler and can be assigned values later when the module is linked or loaded for execution.

For those cases where a specific segment is desired, the following notation may be used (the segment designator is enclosed in double angle brackets):

<<segment>>offset

where "segment" is a constant expression which evaluates to a 7-bit value, and "offset" is a constant expression which evaluates to a 16-bit value. This notation is expanded into a long offset address by the assembler.

LD RO, #DATATABLE + 12 !Add 12 to the address of DATATABLE and load the result into word register RO (non-segmented mode)!

LDL RR2, #ADDR !Load the address of ADDR into register pair RR2 (segmented mode)!

LDL RR2, #<<3>>%1234 !Load the segmented address with segment 3, offset %1234 into register pair RR2 (segmented mode)!

2.4.2 Register Address

In Register addressing mode, the operand value is the contents of the specified general-purpose register. There are four different sizes of registers on the Z8000.

- Word register (16 bits)
 - Byte register (8 bits)
 - Register pair (32 bits)
- Register quadruple (64 bits)

A word register is indicated by an "R" followed by a number from 0 to 15 (decimal). These correspond to the 16 registers of the machine. Either the high or low byte of the first eight registers can be accessed by using the byte register constructs "RH" or "RL" followed by a number from 0 to 7. Any pair of word registers can be accessed as a register pair by using "RR" followed by an even number between 0 and 14. Register quadruples are equivalent to four consecutive word registers and are accessed by the notation "RQ" followed by one of the numbers 0, 4, 8, or 12.

If an odd register number is given with a register pair designator, or a number other than 0, 4, 8, or 12 is given for a register quadruple, an assembly error will result.

In general, the size of a register used in an operation depends on the particular instruction. Byte registers are used with byte instructions, which end with the suffix "B". Word registers are used with word instructions, which have no special suffix. Register pairs are used with long word instructions, which end with the suffix "L". Register quadruples are used only with three instructions (DIVL, EXTSL and MULTL) which use a 64-bit value. An assembly error will occur if the size of a register does not correspond correctly with the particular instruction.

LD R5, #%3FFF !Load register 5 with the hexadecimal value 3FFF! LDB !Load the high order byte of RH3, #%F3 word register 3 with the hexadecimal value F3! !Add the register pairs 2-3 and ADDL RR2, RR4 4-5 and store the result in 2-3! !Multiply the value in register MULTL RO8, RR12 pair 10-11 by the value in register pair 12-13 and store the result in register quadruple 8-9-10-11!

2.4.3 Indirect Register Address

In Indirect Register addressing mode, the operand value is the contents of the location whose address is contained in the specified register. A word register is used to hold the address in nonsegmented mode, while a register pair must be used in segmented mode. Any general-purpose word register (or register pair in segmented mode) can be used except RO or RRO.

Indirect Register addressing mode is also used with the I/O instructions and always indicates a 16-bit I/O address. Any general-purpose word register can be used except RO.

An Indirect Register address is specified by a "commercial at" symbol (@) followed by either a word register or a register pair designator. For Indirect Register addressing mode, a word register is specified by an "R" followed by a number from 1 to 15, and a register pair is specified by a "RR" followed by an even number from 2 to 14 (Section 2.4.2).

| JP | @R2 | !Pass control (jump) to the program memory location addressed by register 2 (nonsegmented mode)! |
|----|-----------|--|
| LD | @R3, R2 | !Load contents of register 2 into location addressed by register 3 (nonsegmented mode)! |
| LD | @RR2, #30 | !Load immediate value 30 into location addressed by register pair 2-3 (segmented mode)! |

2.4.4 Direct Address

The operand value used by the instruction in Direct addressing mode is the contents of the location specified by the address in the instruction. A direct address may be specified as a symbolic name of a memory or I/O location, or an expression which evaluates to an address. In nonsegmented mode or for all I/O instructions, the address is a 16-bit value. In segmented mode, the memory address is either a 16-bit value (short offset) or a 32-bit value (long offset). All assembly-time address expressions are evaluated using 32-bit arithmetic, with only the rightmost 16 bits of the result used for nonsegmented addresses.

LD R10, DATUM

!Load the contents of the location addressed by DATUM into register 10!

LD STRUCT+8, R10

!Load the contents of register 10 into the location addressed by adding 8 to STRUCT!

JP C, %2000

!Jump to location %2000 if the carry flag is set (nonsegmented mode)!

INB RH0, 77

!Input the contents of the I/O location addressed by 77 into RH0!

For segmented addresses, the assembler automatically creates the proper format which includes the segment number and the offset. It is recommended that symbolic names be used wherever possible, since the corresponding segment number and offset for the symbolic name will be automatically managed by the assembler and can be assigned values later when the module is linked or loaded for execution.

For those cases where a specific segment is desired, the following notation may be used (the segment designator is enclosed in double angle brackets):

<<segment>>offset

where "segment" is a constant expression which evaluates to a 7-bit value, and "offset" is a constant expression which evaluates to a 16-bit value. This notation is expanded into a long offset address by the assembler.

To force a short offset address, the segmented address may be enclosed in vertical bars ("|"). In this case, the offset must be in the range 0 to 255, and the final address includes the segment number and short offset in a 16-bit value.

LD R10, |DATUM|

!Load the contents of the location addressed by DATUM (short offset format) into register 10!

LD <<STORAGE>>HEAD, R10 !Load the contents of register 10 into the location addressed by the segment named STORAGE offset by HEAD (long offset format)!

JP |<<3>>%12| Jump to location addressed by segment 3, offset %12 (short offset format)!

2.4.5 Indexed Address

An indexed address consists of a memory address displaced by the contents of a designated word register (the index). This displacement is added to the memory address and the resulting address points to the location whose contents are used by the instruction. In nonsegmented mode, the memory address is specified as an expression which evaluates to a 16-bit value. In segmented mode, the memory address is specified as an expression which evaluates to either a 16-bit value (short offset) or a 32-bit value (long offset). All assembly-time address expressions are evaluated using 32-bit arithmetic, with only the rightmost 16 bits of the result used for nonsegmented addresses. This address is followed by the index, a word register designator enclosed in parentheses. For Indexed Addressing, a word register is specified by an "R" followed by a number from 1 to 15. Any general-purpose word register can be used except RO.

LD R10, TABLE(R3) !Load the contents of the location addressed by TABLE plus the contents of reg-Berraeb al grampos of Losquister 3 into register 10!

LD 240+38(R3), R10 [and !Load the contents of register 10 into the location addressed by 278 plus the contents of register 3 of address, doing doing (nonsegmented mode)!

For segmented addresses, the assembler automatically creates the proper format for the memory address which includes the segment number and the offset. As with Direct Addressing, symbolic names should be used wherever possible so that values can be assigned later when the module is linked or loaded for execution.

For those cases where a specific segment is desired, the following notation may be used (the segment designator is enclosed in double angle brackets):

<<segment>>offset(R)

where "segment" is a constant expression which evaluates to a 7-bit value, "offset" is a constant expression which evaluates to a 16-bit value, and "R" is a word register designator. This notation is expanded into a long offset address by the assembler.

To force a short offset address, the segmented address may be enclosed in vertical bars ("|"). In this case, the offset must be in the range 0 to 255, and the final address includes the segment number and short offset in a 16-bit value.

LD R10, |TABLE|(R3) !Load the contents of the location addressed by TABLE (short offset format) plus the contents of register 3 into register 10!

LD <<STACK>>8(R13), R10 !Load the contents of register 10 into the location addressed by the segment named STACK offset by 8 (long offset format) plus the contents of register 13!

2.4.6 Relative Address

Relative address mode is implied by its instruction. It is used by the Call Relative (CALR), Decrement and Jump If Not Zero (DJNZ), Jump Relative (JR), Load Address Relative (LDAR), and Load Relative (LDR) instructions and is the only mode available to these instructions. The operand, in this case, represents a displacement that is added to the contents of the program counter to form the destination address that is relative to the current instruction. The original contents of the program counter is taken to be the address of the instruction byte following the instruction. The size and range of the displacement depends on the particular instruction, and is described with each instruction in Section 3.

The displacement value can be expressed in two ways. In the first case, the programmer provides a specific displacement in the form "\$+n" where n is a constant expression in the range appropriate for the particular instruction and \$ represents the contents of the program counter at the start of the instruction. The assembler automatically subtracts the value of the address of the following instruction to derive the actual displacement.

JR OV. \$+K !Add value of constant K to program counter and jump to new location if overflow has occurred!

In the second method, the assembler calculates the displacement automatically. The programmer simply specifies an expression which evaluates to a number or a program label as in Direct Addressing. The address specified by the operand must be in the valid range for the instruction, and the assembler automatically subtracts the value of the address of the following instruction to derive the actual displacement.

DJNZ R5, LOOP !Decrement register 5 and jump to LOOP if the result is not zero!

LDR R10, DATA

Load the contents of the location addressed by DATA into register 10!

2.4.7 Based Address

A based address consists of a register that contains the base and a 16-bit displacement. The displacement is added to the base and the resulting address indicates the location whose contents are used by the instruction.

In nonsegmented mode, the base address is held in a word register that is specified by an "R" followed by a number from 1 to 15. Any general-purpose word register can be used except RO. The displacement is specified as an expression which evaluates to a 16-bit value, preceded by a "#" symbol and enclosed in parentheses.

In segmented mode, the segmented base address is held in a register pair that is specified by an "RR" followed by an even number from 2 to 14. Any general-purpose register pair can be used except RRO. The displacement is specified as an expression which evaluates to a 16-bit value, preceded by a "#" symbol and enclosed in parentheses.

LDL RR2, R1(#255) !Load into register pair 2-3 the long word value found in the location resulting from adding 255 to the address in register 1 (nonsegmented mode)!

LD RR4(#%4000), R2 !Load register 2 into the location addressed by adding %4000 to the segmented address found in register pair 4-5 (segmented mode)!

2.4.8 Based Indexed Address

Based Indexed addressing is similar to Based addressing (2.4.7) except that the displacement ("index") as well as the base is held in a register. The contents of the registers are added together to determine the address used in the instruction.

In nonsegmented mode, the base address is held in a word register that is specified by an "R" followed by a number from 1 to 15. The index is held in a word register specified in a similar manner and enclosed in parentheses. Any general-purpose word registers can be used for either the base or index except RO.

In segmented mode, the segmented base address is held in a register pair that is specified by an "RR" followed by an even number from 2 to 14. Any general-purpose register pair can be used except RRO. The index is held in a word register that is specified by an "R" followed by a number from 1 to 15. Any general-purpose word register can be used except RO.

R3, R8(R15)

!Load the value at the location addressed by adding the address in R8 to the displacement in R15 into register 3 (nonsegmented mode)!

LDB RR14(R4), RH2 !Load register km2 income segmented address in RR14 indexed by the value in R4 (segmented mode)!

Section 3 **Assembly Language Instruction Set**

3.1 FUNCTIONAL SUMMARY

Z8000 PLZ/ASM instructions can be divided functionally into nine groups:

- Load and Exchange
- Arithmetic

- Program Control

 Bit Manipulation
- Bit Manipulation
 Rotate and Shift
 Block Transfer and String Manipulation
 - Input/Output
 - CPU Control

The instruction summary shows the instructions belonging to each functional group and the number of operands required for each. The following notation is used:

- Operations with word, byte and long word data operands are listed with their instruction mnemonics grouped together. The suffix "B" designates a byte instruction, the suffix "L" designates a long word instruction, and no suffix designates a word instruction. For some instructions, a data size is either not applicable or depends on the segmentation mode, and thus the instruction mnemonic does not have a suffix to indicate data size.
- "src" is the source operand ("srcl" and "src2" are used to distinguish between two source operands in the same instruction)
- "dst" is the destination operand
- "r" is a register operand, typically used as a counter
- "num" is a number; that is, an immediate value
- "cc" is a condition code (Section 3.2.1)
- "flag" is any combination of the C, Z, S, P, and V status flags
- "int" is any combination of the VI and NVI interrupt control bits

LOAD AND EXCHANGE INSTRUCTIONS

| Instruction | Operand(s) | Name of Instru | ction |
|---------------------|----------------|----------------|----------|
| CLR CLRB | autbA dst losd | Clear | |
| | | | |
| EXB | dst,src | Exchange | |
| | | | |
| LD LDB | dst,src | Load | |
| LDL | | | |
| LDA | dst,src | Load Address | |
| LDAR | dst,src | Load Address I | Relative |
| LDK | dst,src | Load Constant | |
| LDM | dst,src,num | Load Multiple | |
| LDR LDRB LDRL | dst,src | Load Relative | |
| | | | |
| POP POPL | dst,src | Pop | |
| | | | |
| PUSH PUSHL | dst,src | Push | |

ARITHMETIC INSTRUCTIONS

| Instruction | Operand(s) | Name of Instruction |
|---------------------|-----------------------|---------------------|
| ADC ADCB | dst,src | Add with Carry |
| ADD ADDB ADDL | dst,src Complement | Add COMB |
| CP CPB CPL | dst,src | Compare 90 |

ARITHMETIC INSTRUCTIONS (continued)

| Instruction | Operands(s) | Name of Instruction |
|---------------------|----------------|---------------------|
| DAB | dst | Decimal Adjust |
| DEC DECB | dst,src | Decrement |
| DIVL | dst,src | Divide |
| EXTS EXTSB | đst | Extend Sign 100 |
| EXTSL | | |
| INCB | age dst,src | Increment |
| MULT | dst,src | Multiply |
| NEG NEGB | evide dst book | Negate 6 gag |
| SBC SBCB | dst,src | Subtract with Carry |
| SUB SUBB SUBL | dst,src | Subtract |

LOGICAL INSTRUCTIONS

| Instruction | Operand(s) | Name of Instr | uction |
|-------------|------------|---------------|--------|
| | | | |
| AND | dst,src | And | |
| ANDB | | | |
| | | | |
| COM | dst | Complement | |
| COMB | | | |
| | | | |
| OR | dst,src | Ora, Jab | |
| ORB | | | |
| | | | |

LOGICAL INSTRUCTIONS (continued)

| Instruction | Operand(s) | Name of Instruction |
|------------------------|------------|---------------------|
| TEST TESTB TESTL | dst | Test |
| TCC TCCB | cc,dst | Test Condition Code |
| XOR XORB | dst,src | Exclusive Or |

PROGRAM CONTROL INSTRUCTIONS

| Instruction | Operand(s) | Name of Instruction |
|---------------------|------------|--------------------------------|
| CALL | dst | Call Procedure |
| CALR | dst | Call Procedure Relative |
| DJNZ DBJNZ | r,dst | Decrement and Jump if Not Zero |
| IRET | | Interrupt Return |
| JP depoint adpia | cc,dst | Jump |
| JR | cc,dst | Jump Relative |
| RET | cc | Return from Procedure |
| SC IA DIMENYO | src | System Call |

BIT MANIPULATION INSTRUCTIONS

| Instruction | Operand(s) | Name of Instruction | <u> </u> |
|-------------|------------|---------------------|----------|
| BIT BITB | dst,src | Bit Test | |

BIT MANIPULATION INSTRUCTIONS (continued)

| Instruction | Operand(s) | Name of Instruction |
|---------------|------------|---------------------|
| RES RESB | dst,src | Reset Bit |
| SET SETB | dst,src | Set Bit |
| TSET TSETB | dst | Test and Set |

ROTATE AND SHIFT INSTRUCTIONS

| Instruction | Operand(s) | Name of Instruction |
|---------------------|------------|-------------------------------|
| RL RLB | dst,src | Rotate Left |
| RLC RLCB | dst,src | Rotate Left through Carry |
| RLDB | dst,src | Rotate Left Digit |
| RR RRB | dst,src | Rotate Right TEGI |
| RRC RRCB | dst,src | Rotate Right through Carry |
| RRDB | dst,src | Rotate Right Digit |
| SDA SDAB SDAL | dst,src | Shift Dynamic Arithmetic |
| SDL SDLB SDLL | dst,src | Shift Dynamic Logical |
| SLA SLAB SLAL | dst,src | Shift Left Arithmetic |

ROTATE AND SHIFT INSTRUCTIONS (continued)

| <u>Instruction</u> <u>Op</u> | perand(s) | Name of Instruction |
|------------------------------|-----------|------------------------|
| SLL ds SLLB SLLL | | Shift Left Logical |
| SRA ds SRAB SRAL | | Shift Right Arithmetic |
| SRL ds SRLB SRLL | | Shift Right Logical |

BLOCK TRANSFER AND STRING MANIPULATION INSTRUCTIONS

| Instruction | Operand(s) | Name of Instruction |
|-----------------|--------------------|--------------------------------------|
| CPD CPDB | dst,src,r,cc | Compare and Decrement |
| CPDR CPDRB | dst,src,r,cc | Compare, Decrement and Repeat |
| CPI CPIB | dst,src,r,cc | Compare and Increment |
| CPIR CPIRB | dst,src,r,cc | Compare, Increment and Repeat |
| CPSD CPSDB | dst,src,r,cc | Compare String and Decrement |
| CPSDR CPSDRB | dst,src,r,cc TUGTO | Compare String, Decrement and Repeat |
| CPSI CPSIB | dst,src,r,cc | Compare String and Increment |
| CPSIR CPSIRB | dst,src,r,cc | Compare String, Increment and Repeat |
| LDD LDDB | dst,src,r | Load and Decrement |

BLOCK TRANSFER AND STRING MANIPULATION INSTRUCTIONS (continued)

| Instruction | Operand(s) | Name of Instruction |
|----------------------------------|-------------|---------------------------------------|
| LDDR LDDRB | dst,src,r | Load, Decrement and Repeat |
| LDI LDIB _{IA MAD} IR | dst,src,r | Load and Increment |
| LDIR LDIRB | dst,src,r | Load, Increment and Repeat |
| TRDB | dst,src,r | Translate and Decrement |
| TRDRB | dst,src,r | Translate, Decrement and Repeat |
| TRIB | dst,src,r | Translate and Increment |
| TRIRB | dst,src,r | Translate, Increment and Repeat |
| TRTDB | srcl,src2,r | Translate, Test and Decrement |
| TRTDRB | srcl,src2,r | Translate, Test, Decrement and Repeat |
| TRTIB | srcl,src2,r | Translate, Test and Increment |
| TRTIRB | srcl,src2,r | Translate, Test, Increment and Repeat |

INPUT/OUTPUT INSTRUCTIONS

| Instruction | Operand(s) | Name of | Instruction |
|-------------|------------|---------|--------------|
| IN INB | dst,src | Input | |
| IND INDB | dst,src,r | Input a | nd Decrement |

INPUT/OUTPUT INSTRUCTIONS (continued)

| Instructions | Operand(s) | Name of Instruction |
|------------------|------------|--------------------------------------|
| INDR HIGHER | dst,src,r | Input, Decrement and Repeat |
| INIB TO THE | dst,src,r | Input and Increment |
| INIR MAGNADINIRB | dst,src,r | Input, Increment and Repeat |
| OTDR OTDRB | dst,src,r | Output, Decrement and Repeat |
| OTIR OTIRB | dst,src,r | Output, Increment and Repeat |
| OUT OUTB | dst,src | Output |
| OUTD OUTDB | dst,src,r | Output and Decrement |
| OUTI OUTIB | dst,src,r | Output and Increment |
| SIN SINB | dst,src | Special Input |
| SIND SINDB | dst,src,r | Special Input and Decrement |
| SINDR SINDRB | | Special Input, Decrement and Repeat |
| SINI SINIB | dst,src,r | Special Input and Increment |
| SINIR SINIRB | dst,src,r | Special Input, Increment and Repeat |
| SOTDR SOTDRB | dst,src,r | Special Output, Decrement and Repeat |
| SOTIR SOTIRB | dst,src,r | Special Output, Increment and Repeat |

INPUT/OUTPUT INSTRUCTIONS (continued)

| Instruction | Operand(s) | Name of Instruction |
|--------------|------------|---------------------------------|
| SOUT | dst,src | Special Output |
| SOUTD SOUTDB | dst,src,r | Special Output and Decrement |
| SOUTI SOUTIB | dst,src,r | Special Output and Increment |

CPU CONTROL INSTRUCTIONS

| Instruction | Operand(s) | Name of Instruction |
|-----------------|------------|------------------------------------|
| COMFLG | flag | Complement Flag |
| II EI | int int | Disable Interrupt Enable Interrupt |
| HALT DOLL BOS | | Halt |
| LDCTL LDCTLB | dst,src | Load Control Register |
| LDPS | src | Load Program Status |
| MBIT | | Multi-Micro Bit Test |
| MREQ | dst | Multi-Micro Request |
| MRES | | Multi-Micro Reset |
| | | Multi-Micro Set |
| NOP | | No Operation |
| RESFLG | flag | Reset Flag |
| SETFLG | flag | Set Flag |
| | | |

The following set of instructions are privileged; that is, they can only be executed in System mode:

| DI | INIB | OTDR | SIN | SOTDR |
|-------|-------|-------|--------|--------|
| EI | INIR | OTDRB | SINB | SOTDRB |
| HALT | INIRB | OTIR | SIND | SOTIR |
| IN | IRET | OTIRB | SINDB | SOTIRB |
| INB | LDCTL | OUT | SINDR | SOUT |
| IND | LDPS | OUTB | SINDRB | SOUTB |
| INDB | MBIT | OUTD | SINI | SOUTD |
| INDR | MREQ | OUTDB | SINIB | SOUTDB |
| INDRB | MRES | OUTI | SINIR | SOUTI |
| INI | MSET | OUTIB | SINIRB | SOUTIB |

3.2 NOTATION AND BINARY ENCODING

3.2.1 Operand Notation

Operands are represented by a notational shorthand in the detailed instruction descriptions that make up the rest of this section. The notation for operands and their actual assembly language syntax are as follows:

| Notation | Address Mode | Actual Operand |
|----------|--------------------------|---|
| IM | Immediate Data | #expression |
| R | Register | $\frac{Rn}{RHn}$, where n=0-15; $\frac{RHn}{n=0-7}$, $\frac{RRn}{n=0}$, where n=0,2,4,14; $\frac{RQn}{n=0}$, where n=0,4,8,12 |
| IR | Indirect Register | <pre>@Rn, where n=1-15 for nonsegmented mode; @RRn, where n=2,4,614 for segmented mode</pre> |
| DA | Direct Address | expression |
| Х | Indexed Address | $\frac{\text{expression(Rn)}}{\text{n=1-15}}, \text{ where}$ |
| RA | Relative Address | expression |
| ВА | Based Address | $\frac{\text{Rn}(\text{\#expression})}{\text{n=1-15}}$ for nonsegmented mode; $\frac{\text{RRn}(\text{\#expression})}{\text{n=2,4,614}}$ for segmented mode |
| вх | Based Indexed Address | Rn(Rn), where n=1-15 for nonsegmented mode; RRn(Rm), where n=2,4,614 and m=1-15 for segmented mode |

The control registers (CTLR) are:

| FLAGS | Status Flags |
|---------|-------------------------------------|
| FCW | Flags and Control Word |
| REFRESH | Refresh |
| PSAP | Program Status Area Pointer* |
| PSAPSEG | Program Status Area Pointer Segment |
| PSAPOFF | Program Status Area Pointer Offset |
| NSP | Normal Stack Pointer* |
| NSPSEG | Normal Stack Pointer Segment |
| NSPOFF | Normal Stack Pointer Offset |

^{*} For the nonsegmented Z8002, PSAP is equivalent to PSAPOFF and NSP is equivalent to NSPOFF.

The status flags are:

| С | Carry flag |
|---|----------------------|
| Z | Zero flag |
| S | Sign flag |
| P | Parity flag |
| V | Overflow flag |
| D | Decimal-adjust flag* |
| Н | Half-carry flag* |

^{*} These flags may not be specified in assembly language statements.

The interrupt control bits are:

| VI | Vectored Interrupt |
|-----|------------------------|
| NVI | Non-Vectored Interrupt |

The binary values used in the instruction format encoding for the general-purpose registers are:

| | Reg | ister | | Binary |
|-----|-----|----------|------------|--------------|
| RQ0 | RR0 | RO R1 | RHO RH1 | 0000 |
| | RR2 | R2 R3 | RH2 RH3 | 0010 0011 |
| RQ4 | RR4 | R4 R5 | RH4 RH5 | 0100 0101 |

| | Regi | ster | | Binary |
|------|------|------------|------------|--------------|
| | RR6 | R6 R7 | RH6 RH7 | 0110 0111 |
| RQ8 | RR8 | R8 R9 | RLO RL1 | 1000 1001 |
| | RR10 | R10 R11 | RL2 RL3 | 1010 1011 |
| RQ12 | RR12 | R12 R13 | RL4 RL5 | 1100 1101 |
| | RR14 | R14 R15 | RL6 RL7 | 1110 |

The condition codes and the flag settings they represent are:

| Code | Meaning | Flag Settings | Binary |
|---------|---------------|---------------------|----------|
| | Always false | lags are: | 0000 |
| (blank) | Always true | | 1000 |
| Z | Zero | z=1 | 0110 |
| NZ | Not zero | z = 0 | 1110 |
| C | Carry | C=1 | 0111 |
| NC | No carry | C=0 | 1111 |
| PL | Plus | S=0 | 1101 |
| MI | Minus | S=1 | 0101 |
| NE | Not equal | z = 0 | 1110 |
| EO | Equal | Z=1 | 0110 |
| OV | Overflow | V=1 | 0100 |
| NOV | No overflow | V = 0 | 1100 |
| PE | Parity even | P=1 | 0100 |
| PO | Parity odd | P=0 | 1100 |
| GE | Greater than | (S XOR V) = 0 | 1001 |
| | or equal | | |
| LT | Less than | (S XOR V) = 1 | 0001 |
| GT | Greater than | (Z OR (S XOR V)) = | 0 1010 |
| LE | Less than or | (Z OR (S XOR V)) = | 1 0010 |
| | equal | | |
| UGE | Unsigned | C = 0 | 1111 |
| | greater than | | |
| | or equal | | |
| ULT | Unsigned | C=1 | 0111 |
| | less than | | |
| UGT | Unsigned | ((C=0) AND (Z=0)) = | = 1 1011 |
| | greater than | | |
| ULE | Unsigned less | (C OR Z) = 1 | 0011 |
| | than or equal | | |

Note that some of the condition codes correspond to identical flag settings: i.e., Z-EQ, NZ-NE, C-ULT, NC-UGE, PE-OV, PO-NOV.

3.2.2 Instruction Format Encoding

The binary encoding of each Z8000 instruction follows one of nine basic formats. Each format determines the grouping of bits into various instruction "fields", the meaning of which depends on the instruction. The following symbols represent these fields:

| Symbol . | Field Size | Value |
|--------------|------------|---|
| mode | 2 bits | Addressing mode specifier |
| src | Varies | Source operand address or immediate value |
| dst | Varies | Destination operand address |
| r | 4 bits | Register counter address |
| num | 4 bits | Number (immediate value) |
| СС | 4 bits | Condition code |
| W/B | l bit | Word or byte specifier |
| х | 4 bits | Index register address for Based Indexed Addressing mode |
| displacement | t Varies | Displacement for Relative or Based Addressing modes |

All other bit fields are referred to as operation code (opcode) fields because they determine the operation of the instruction.

In addition to the various fields within an instruction, address information for the data operands may be appended to the basic instruction encoding. The size and format of this additional address information depends on the particular address mode, as well as the data operand size in the case of Immediate Data, or the segmentation mode in the case of Direct Address or Indexed Address.

The standard formats for address information are illustrated by the following examples. Nonstandard formats are described in the detailed description for each instruction (Section 3.3). The following examples use Addition (ADD), Load (LD), and Load Relative (LDR) instructions for illustration purposes, although the address information formats apply to all instructions which require additional address information.

Register (R)

 $\begin{array}{ccc} & \underline{\text{dst}} & \underline{\text{src}} \\ \text{ADD} & \overline{\text{R}}, & \overline{\text{R}} \end{array}$

mode opcode W/B src dst

 $\{\text{mode} = 10\}$

Example:

ADD R2, R3

1 0 0 0 0 0 0 1 0 0 1 1 0 0 1 0

Immediate Data -- Word (IM)

 $\begin{array}{ccc} & \underline{\text{dst}} & \underline{\text{src}} \\ \text{ADD} & \overline{R}, & \underline{\text{IM}} \end{array}$

| mode | opcode | W/B | src | dst |
|------|--------|------|-----|-------|
| G AT | | data | | T. T. |

 $\{\text{mode} = 00, \text{ src field} = 0\}$

Example:

ADD R2, #%1234

| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 1 | 0 | 1 | 0 | 0 |

Immediate Data -- Byte (IM)

ADDB $\frac{\text{dst}}{R_{\star}}$ $\frac{\text{src}}{\text{IM}}$

| mode | opcode | W/B | src | dst | |
|------|--------|-----|-----|------|--|
| | data | | | data | |

{mode = 00, src field = 0, Byte Immediate Data is <u>duplicated</u> in each half of the second instruction word}

Example:

ADDB RH2, #%12

| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | | | | | | | | 0 | | | | | | | |

Immediate Data -- Long (IM) 10 danda balbangan -- sasaba basala

 $\begin{array}{cccc} & \underline{\text{dst}} & \underline{\text{src}} \\ \text{ADDL} & R, & \underline{\text{IM}} \end{array}$

| mode | opcode | src | dst |
|-------|----------|-----------|-----|
| ngeit | data (hi | gh order) | |
| | data (lo | ow order) | |

 $\{\text{mode} = 00, \text{ src field} = 0\}$

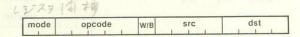
Example:

ADDL RR2, #%12345678

| 0 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |
|---|---|---|----|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 1 | 0 | 1 | 0 | 0 |
| 0 | 1 | 0 | 1- | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 0 |

Indirect Register (IR)

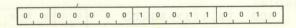
 $\begin{array}{ccc} & \underline{\text{dst}} & \underline{\text{src}} \\ \text{ADD} & \overline{\text{R}}, & \overline{\text{IR}} \end{array}$



 $\{\text{mode} = 00, \text{ src field } \langle \rangle 0\}$

Example:

ADD R2, @R3



Direct Address -- Nonsegmented (DA)

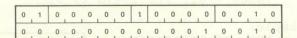
 $\begin{array}{ccc} & \underline{\text{dst}} & \underline{\text{src}} \\ \text{ADD} & \overline{\text{R}}, & \overline{\text{DA}} \end{array}$

| mode | opcode | W/B | src | dst |
|------|--------|--------|-----|-----|
| | | addres | | |

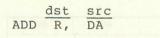
 $\{\text{mode} = 01, \text{ src field} = 0\}$

Example:

ADD R2, %0012



Direct Address -- Segmented Short Offset (DA)



| mode | opcode | W/B | src | dst |
|------|---------|-----|-----|------|
| 0 | segment | | of | fset |

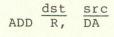
 $\{\text{mode} = 01, \text{ src field} = 0\}$

Example:

ADD R2, | << 3>>%0012|

| 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | | | | 0 | | | | | | 1 | 0 |

Direct Address -- Segmented Long Offset (DA)



| mode | opcode | W/B | | s | rc | | | d | st | |
|------|---------|------|-----|---|----|----|---|---|----|---|
| 1 | segment | | 0 | 0 | 0 | 0 | 0 | 0 | 0 | C |
| | | offs | set | | | N/ | | | | |

{mode = 01, src field = 0}

Example:

ADD R2, <<3>>%0012

| 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 |

Indexed Address -- Nonsegmented (X)

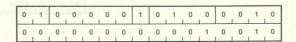
 $\begin{array}{ccc} & \underline{\text{dst}} & \underline{\text{src}} \\ \text{ADD} & \overline{R}, & \underline{X} \end{array}$

| node | opcode | W/B | src | dst |
|------|---------|-----|-------|-----|
| 1 | 1 1 1 1 | 1 | 1 1 1 | 1 1 |

{mode = 01, src field <> 0}

Example:

ADD R2,%0012(R4)



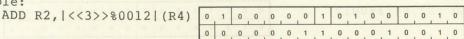
Indexed Address -- Segmented Short Offset (X)

ADD $\frac{\text{dst}}{R}$, $\frac{\text{src}}{X}$

| mode | opcode | W/B | src | dst |
|------|---------|-----|-----|-----|
| 0 | segment | | off | set |

{mode = 01, src field <> 0}

Example:



Indexed Address -- Segmented Long Offset (X)

| | dst | src |
|-----|-----|-----|
| ADD | R, | X |

| mode | opcode | W/B | | s | rc | 18 | | d | st | di |
|------|---------|------|----|---|----|----|---|---|----|----|
| 10 0 | segment | | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| | | offs | et | | | | | | | |

Example:

ADD R2, <<3>>%0012(R4)

| 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 |

Relative Address (RA)

 $LDR = \frac{dst}{R}, \frac{src}{RA}$

| 0 | pcode | W/B | 0 | 0 | 0 | 0 | dst |
|---|-------|---------|-----|----|---|---|-----|
| | 201 | displac | eme | nt | | | |

Example:

LDR R2,\$+10

| 0 | 0 | 1 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | | | | | | | | | | | 0 |

{The assembler automatically calculates the relative address displacement (=6) as the given value (10 bytes after the start of the LDR instruction) minus the start of the following instruction (4 bytes after the start of the LDR instruction)}

Based Address (BA)

LD $\frac{\text{dst}}{R}$, $\frac{\text{src}}{BA}$

opcode W/B src dst displacement

{src field <> 0}

Example:

LD R2, R4(#%0012)

| 0 | 0 | 1 | 1 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 |

Based Indexed Address (BX) + 20110 prod Bestrempo2 -- 2

LD $\frac{\text{dst}}{R}$, $\frac{\text{src}}{BX}$

| go | 0 | pcoc | le | , | W/B | | S | rc | | | ds | st | 10 |
|-----|---|------|----|---|-----|---|---|----|---|---|----|----|----|
| 0 0 | 0 | .0 | | X | | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

{src field <> 0}

Example:

LD R2, R4 (R5)

| 0 | 11 | 1 | 1 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 0 |
|---|----|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

There are several instructions whose source operand is Immediate Data and destination operand is either Direct Address or Indexed Address. In this case, the Immediate Data follows the address information (DA or X) in the instruction encoding. The following example uses the nonsegmented DA format

 $LD \quad \frac{dst}{DA}, \quad \frac{src}{IM}$

Example:

LD %0012, #%1234

| mo | de | | 0 | pcod | de | | W/B | | d | st | | | opc | ode | |
|-----|-----|---|---|------|-----|---|------|-----|---|-----|-----|---|-----|-----|-----|
| | | | | 1 | | | addı | ess | | n i | i p | | a a | - | |
| -1 | | | 1 | | | | 1 1 | | | | | _ | | | _ |
| - | | | | | | | dat | ta | | | | | | | |
| | | _ | _ | - | | | | _ | | | _ | - | _ | - | _ |
| | | | | | | | | | | | | | | | |
| 0 . | 1 | 0 | 0 | 1 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 |
| 0 1 | 1 0 | 0 | 0 | 1 0 | 1 0 | 0 | 1 0 | 0 | 0 | 0 | 0 | 0 | 1 0 | 0 | 1 0 |

The nine basic formats, Fl through F9, determine the size of the instruction, which also depends on the addressing mode. The following table can be used to determine the number of bytes in a particular instruction. The addressing mode (including the data operand size and whether segmented or nonsegmented mode is used) specifies the row in the table, while the format number specifies the column.

TABLE 3-1. NUMBER OF BYTES IN INSTRUCTIONS

| MODE | | | | FC | RMAT | | | | |
|----------------|------|----|------|------|------|----|-----|------|------|
| dst or src | Fl | F2 | F3 | F4 | F5 | F6 | F7 | F8 | F9 |
| R, IR | 2 | 2 | ego | 1938 | 4 | 4 | 2 | 2 | |
| IM (Word/Byte) | 4 | 4 | 2 | | | | | | 2 |
| IM (Long) | 6 | 6 | | | dann | | | | |
| DA,X (NS/SS) | 4 | 4 | | | 6 | 6 | 4 | | |
| DA,X (SL) | 6 | 6 | | | 8 | 8 | | | |
| RA | | | 2 | 4 | 6000 | | | | |
| BA | | | | 4 | | | | | |
| BX | 5774 | To | 9075 | 4 | 6778 | | 100 | 1701 | 3277 |

NS = Nonsegmented, SS = Segmented Short Offset, SL = Segmented Long Offset

3.2.3 Operation Notation

The description of each instruction's operation includes a shorthand summary. The following symbols are used in the operation summary.

| Symbol | Meaning |
|-------------|---|
| src | Source operand |
| dst | Destination operand |
| r s s s | Register operand (typically used as a counter) |
| num | Number, that is, an immediate value |
| cc | Condition code |
| instruction | Binary value of the instruction itself |
| Rn | Register number "n" |
| PC | Program Counter register |
| PS | Program Status register |
| SP | Processor stack pointer register (R15 if nonsegmented, RR14 if segmented) |
| <- | Becomes (assignment of a value) |
| @n | <pre>Indirect; that is, the value found at the address contained in "n"</pre> |
| n1,n2. | Concatenation; that is, both "nl" and "n2" are operated on together |
| fyn | Operand addressed by the Boolean OR of the instruction field "f" and the value "n" (e.g., ROV1 is R1) |

Symbol Meaning

operand(n) Bit "n" of operand

operand (msb) Most significant bit of operand

operand(n1:n2) Bits "n1" through "n2"

of operand

operand whose address is

contained in the source operand offset by the destination operand.

AUTOINCREMENT The address of the operand is

automatically incremented by the size of the operand in bytes (the address is always contained in a

register)

AUTODECREMENT The address of the operand is

automatically decremented by the size of the operand in bytes (the address is always contained in

a register)

3.3 ASSEMBLY LANGUAGE INSTRUCTIONS

In the remainder of this chapter, Z8000 assembly language instructions are described in detail in alphabetic order. Each description includes:

- The name of the instruction.
- The assembly language statement format(s) including the required operands and valid addressing modes (see section 3.2.1).
- The binary instruction format(s) including the instruction fields and the format class (see section 3.2.2). If the instruction has a "mode" field, then the valid addressing modes for that format are listed to the right of the binary encoding. If the instruction has a "W/B" field, then a "l" bit indicates a word operation and a "0" bit indicates a byte operation. The format class appears in parentheses above the binary encoding and consists of two numbers (Fn.m). "Fn" specifies the basic format class Fl through F9 and can be used with Table 3-1 to determine the number of bytes for a particular instruction. The "m" portion of the format number specifies the sub-class (see Appendix A).
- The operation performed by the instruction including a summary description (see section 3.2.3) and a detailed description.
- The status flags affected by the instruction.
- The number of machine cycles used to execute the instruction.
- Special notes for some instructions concerning nonstandard instruction field encodings, hardware related functions, operation restrictions, and other architectural or implementation details.
- A short assembly language example showing the use of the instruction. All examples assume the nonsegmented mode unless otherwise specified.

ADC dst,src ADCB

dst: R src: R

INSTRUCTION FORMAT: (F2.1)

W/B: mode 1 1 0 1 0 W/B mode dst src 10

OPERATION: dst <- dst + src + C

The source operand, along with the setting of the carry flag, is added to the destination operand and the sum is stored in the destination. The contents of the source are not affected. Two's complement addition is performed. In multiple precision arithmetic, this instruction permits the carry from the addition of low-order operands to be carried into the addition of high-order operands.

FLAGS: C: Set if there is a carry from the most significant bit of the result; cleared otherwise

Z: Set if the result is zero; cleared otherwise

S: Set if the result is negative; cleared otherwise

V: Set if arithmetic overflow occurs, that is, if both operands were of the same sign and the result is of the opposite sign; cleared otherwise

D: ADC - unaffected; ADCB - cleared

H: ADC - unaffected; ADCB - set if there is a carry from the most significant bit of the low-order four bits of the result; cleared otherwise

CYCLES: Word/Byte

Long addition may be done with the following EXAMPLE: instruction sequence, assuming RO, Rl contain one operand and R2,R3 contain the other operand:

> ADD R1,R3 !add low-order words! ADC RO, R2 !add carry and high-order words!

If RO contains %0000, Rl contains %FFFF, R2 contains %4320 and R3 contains %0001, then the above two instructions leave the value %4321 in RO and %0000 in Rl.



ADD dst, src ADDB ADDL

dst: R

src: R, IM, IR, DA, X

INSTRUCTION FORMAT: (F2.1)

| W/B: | mode | 0 | 0 | 0 | 0 | 0 | W/B | src | dst |
|------|------|---|---|---|---|---|-----|-----|-----|
| L: | mode | 0 | 1 | 0 | 1 | 1 | 0 | src | dst |

dst src mode R 10 R R IM (src field=0) 00 00 R IR (src field<>0) 01 R DA (src field=0) 01 R X (src field<>0)

OPERATION: dst <- dst + src

The source operand is added to the destination operand and the sum is stored in the destination. The contents of the source are not affected. Two's complement addition is performed.

- FLAGS: C: Set if there is a carry from the most significant bit of the result; cleared otherwise
 - Z: Set if the result is zero; cleared otherwise
 - S: Set if the result is negative; cleared otherwise
 - V: Set if arithmetic overflow occurs, that is, if both operands were of the same sign and the result is of the opposite sign; cleared otherwise
 - D: ADD, ADDL unaffected; ADDB cleared
 - H: ADD, ADDL unaffected; ADDB set if there is a carry from the most significant bit of the low-order four bits of the result; cleared otherwise

CYCLES:

| src | Wo | rd/b | yte | | Long | | | |
|-----|----|------|----------|----|------|---------|--|--|
| | NS | SS | SL | NS | SS | SL | | |
| R | 4 | | | 8 | | | | |
| IM | 7 | | | 14 | / | ALT THE | | |
| IR | 7 | 0 |) Hamili | 14 | one. | | | |
| DA | 9 | 10 | 12 | 15 | 16 | 18 | | |
| X | 10 | 10 | 13 | 16 | 16 | 19 | | |

EXAMPLE: If register R2 contains %0344 and the word at the location named AUGEND contains %0011, the statement

ADD R2, AUGEND

will leave the value %0355 in R2.

dst,src AND ANDB

dst: R

mode dst

src: R, IM, IR, DA, X

INSTRUCTION FORMAT: (F2.1)

| W/B: | mode | 0 | 0 | 0 | 1 | 1 | W/B | src | dst |
|------|------|---|---|---|---|---|-----|-----|-----|
|------|------|---|---|---|---|---|-----|-----|-----|

10 R R 00 R IM (src field=0) 00 R TR (src field<>0) 01 R DA (src field=0) X (src field<>0) 0.1 R

src

OPERATION:

dst <- dst AND src

The source operand is logically ANDed with the destination operand and the result is stored in the destination. The contents of the source are not affected. The AND operation results in a one bit being stored whenever the corresponding bits in the two operands are both ones; otherwise, a zero bit is stored.

FLAGS: C: Unaffected

Z: Set if the result is zero; cleared otherwise

S: Set if the most significant bit of the result is set; is even; cleared otherwise

P: AND - unaffected; ANDB - set if parity of the result is even; cleared otherwise

D: Unaffected

H: Unaffected

CYCLES:

| src | Wo | rd/B | yte |
|-----|----|------|-----|
| | NS | SS | SL |
| R | 4 | | |
| IM | 7 | | |
| IR | 7 | | |
| DA | 9 | 10 | 12 |
| X | 10 | 10 | 13 |

EXAMPLE:

If register RL3 contains %C3 (Il000011) and the source operand is the immediate value %7B (01111011), the statement

ANDB RL3, #%7B

will leave the value %43 (01000011) in RL3.

dst: R, IR, DA, X src: R, IM dst,src BIT BITB INSTRUCTION FORMAT: (F1.2)mode dst src W/B: mode TM 10 R 1 0 0 1 1 W/B 00 IM (dst field<>0) IR 01 IM (dst field=0) DA 01 X IM (dst field<>0) (F6.3)dst src 0 1 1 W/B 0 0 0 src R W/B: dst 0 0 0 0 0 0 0 0 0 0 0 OPERATION: Z <- NOT dst(src)

Tests the specified bit within the destination operand, and sets the Z flag if the specified bit is 0; otherwise it clears the Z flag. The contents of the destination are not affected. The source (the bit number) can be specified as either an immediate value, or as a word register which contains the value. In the second case, the destination operand must be a register, and the source operand must be R0 through R7 for BITB, or R0 through R15 for BIT. The bit number is a value from 0 to 7 for BITB, or 0 to 15 for BIT, with 0 indicating the least significant bit.

FLAGS: C: Unaffected

Z: Set if specified bit is zero; cleared otherwise

S: Unaffected V: Unaffected D: Unaffected H: Unaffected

CYCLES: Word/Byte dst src NS SS SL R IM 4 8 IR IM 11 13 DA TM 10 X IM 11 11 14 R 10 R

NOTE:

Only the lower four bits of the source operand are used to specify the bit number for BIT, while only the lower three bits of the source operand are used with BITB. When the source operand is an immediate value, the "src field" in the instruction format encoding contains the bit number in the lowest four bits for BIT, or the lowest three bits for BITB.

EXAMPLE:

If register RH2 contains %B2 (10110010), the statement

BITB RH2, #0 - 929

will leave the Z flag set.

Call Procedure

CALL dst was add do dst: IR, DA, X

INSTRUCTION FORMAT: (F1.1)

mode dst 00 IR

mode 0 1 1 1 1 1

01 DA (dst field=0) 01 X (dst field<>0)

OPERATION: Nonsegmented Segmented SP <- SP - 4 SP <- SP - 2 @SP <- PC @SP <- PC PC <- dst PC <- dst

> The current contents of the program counter (PC) are pushed onto the top of the processor stack. The stack pointer used is R15 if nonsegmented, or RR14 if segmented. (The program counter value used is the address of the first instruction byte following the CALL instruction.) The specified destination address is then loaded into the PC and points to the first instruction of a procedure.

At the end of the procedure a RET instruction can be used to return to original program flow. RET pops the top of the processor stack back into the PC.

FLAGS:

No flags affected

CYCLES:

dst Address SS SL NS 10 15 IR 18 20 DA 12 18 21 13 X

EXAMPLE:

In nonsegmented mode, if the contents of the program counter are %1000 and the contents of the stack pointer (R15) are %3002, the statement

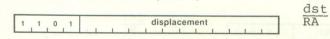
CALL %2520

causes the stack pointer to be decremented to %3000, the value %1004 (the address following the CALL instruction with direct address mode specified) to be loaded into the word at location %3000, and the program counter to be loaded with the value %2520. The program counter now points to the address of the first instruction in the procedure to be executed.

CALR dst

dst: RA

INSTRUCTION FORMAT: (F3.4)



OPERATION:

Nonsegmented
SP <- SP - 2
@SP <- PC
PC <- PC - (2*disp)

Segmented
SP <- SP - 4
@SP <- PC
PC <- PC - (2*disp)

The current contents of the program counter (PC) are pushed onto the top of the processor stack. The stack pointer used is R15 if nonsegmented, or RR14 if segmented. (The program counter value used is the address of the first instruction byte following the CALR instruction.) The destination address is calculated and then loaded into the PC and points to the first instruction of a procedure.

At the end of the procedure a RET instruction can be used to return to the original program flow. RET pops the top of the processor stack back into the PC.

FLAGS:

No flags affected

CYCLES:

Address NS SS SL 10 -- 15

NOTE:

The relative addressing mode is calculated by doubling the displacement in the instruction, then subtracting this value from the updated value of the PC to derive the destination address. The updated PC value is taken to be the address of the instruction byte following the CALR instruction, while the displacement is a 12-bit signed value in the range -2048 to +2047. Thus, the destination address must be in the range -4092 to +4098 bytes from the start of the CALR instruction. In the segmented mode, the PC segment number is not affected. The assembler automatically calculates the displacement by subtracting the PC value of the following instruction from the address given by the programmer.

EXAMPLE:

CALR PROC

!Procedure PROC is called! !Control eventually returns to here!

CLR dst

dst: R, IR, DA, X

INSTRUCTION FORMAT: (F1.1)

W/B: mode 0 0 1 1 0 W/B dst 1 0 0 0 1 R 01 DA (dst field=0) 0 1 X (dst field<>0)

OPERATION:

dst <- 0

The destination is cleared to zero.

FLAGS:

No flags affected

CYCLES:

Word/Byte dst NS SS SL R 7 8 TR 11 12 14 DA 12 12 15 X

EXAMPLE:

If the word at location %ABBA contains 13, the statement

CLR %ABBA

will leave the value 0 in the word at location %ABBA.

COM dst COMB

dst: R, IR, DA, X

INSTRUCTION FORMAT: (F1.1)

mode dst 10 R W/B: mode 0 0 1 1 0 W/B 0 0 0 0 00 IR 01 DA (dst field=0)

01 X (dst field<>0)

OPERATION: dst <- NOT dst

The contents of the destination are complemented (one's complement); all one bits are changed to zero, and viceversa.

FLAGS: C: Unaffected

Z: Set if the result is zero; cleared otherwise

S: Set if the most significant bit of the result is set;

cleared otherwise

P: COM - unaffected; COMB - set if parity of the result

is even; cleared otherwise

D: Unaffected H: Unaffected

Word/Byte CYCLES: dst NS SS R 7 IR 12 18 (1=) isa ai pality and the DA 15 16 16 16 19 X

EXAMPLE: If register Rl contains %2552 (0010010101010010),

the statement

COM R1

will leave the value %DAAD (1101101010101101) in R1.

COMFLG Complement Flag

COMFLG flag

flag: C, Z, S, P, V

INSTRUCTION FORMAT: (F9.1)

1 0 0 0 1 1 0 1 C Z S P/V 0 1 0 1

FLAGS(4:7) <- FLAGS(4:7) XOR instruction(4:7) OPERATION:

Any combination of the C, Z, S, P or V flags are complemented (each one bit is changed to zero, and vice-versa) if the corresponding bit in the instruction is one. If the corresponding bit in the instruction is zero, the flag will not be affected. All other bits in the FLAGS register are unaffected. Note that the P and V flags are represented by the same bit. There may be one, two, three or four operands in the assembly language statement, in any order.

C: Complemented if specified; unaffected otherwise

Z: Complemented if specified; unaffected otherwise

S: Complemented if specified; unaffected otherwise P/V: Complemented if specified; unaffected otherwise

D: Unaffected H: Undefined

CYCLES:

If the C, Z, and S flags are all clear (=0), EXAMPLE: and the P flag is set (=1), the statement

COMFLG P, S, Z, C

will leave the C, Z, and S flags set (=1), and the P flag cleared (=0).

CP dst,src dst: R, IR, DA, X src: R, IM, IR, DA, X CPB CPL INSTRUCTION FORMAT: (F2.1)dst mode src R 10 R W/B: mode 0 0 1 0 1 W/B src R IM (src field=0) 00 IR (src field<>0) R 00 dst L: mode 0 1 0 0 0 0 DA (src field=0) 01 R X (src field<>0) 01 R (F5.1)dst mode Src TM 00 TR W/B: mode 0 W/B dst 0 0 1 1 DA IM (dst field=0) 01 TM (dst field<>0) 0.7 X

OPERATION: dst - src

The source operand is compared to (subtracted from) the destination operand, and the appropriate flags set accordingly, which may then be used for arithmetic and logical conditional jumps. Both operands are unaffected, with the only action being the setting of the flags. Subtraction is performed by adding the two's complement of the source operand to the destination operand.

FLAGS:

- C: Cleared if there is a carry from the most significant bit of the result; set otherwise, indicating a "borrow"
- Z: Set if the result is zero; cleared otherwise
- S: Set if the result is negative; cleared otherwise
- V: Set if arithmetic overflow occurs, that is, if both operands were of opposite signs and the sign of the result is the same as the sign of the source; cleared otherwise
- D: Unaffected
- H: Unaffected

| | - | | - | - | - | - | |
|-----|---|---|---|----|----|---|--|
| - 6 | | V | (| T. | 1. | 5 | |
| | | | | | | | |

| dst | src | W | ord/E | Byte | | Long | | |
|-----|-----|----|-------|------|----|------|----|--|
| 400 | | NS | SS | SL | NS | SS | SL | |
| R | R | 4 | | | 8 | | | |
| R | IM | 7 | | | 14 | | | |
| R | IR | 7 | | | 14 | | | |
| R | DA | 9 | 10 | 12 | 15 | 16 | 18 | |
| R | X | 10 | 10 | 13 | 16 | 16 | 19 | |
| IR | IM | 11 | | | | | | |
| DA | MI | 14 | 15 | 17 | | | | |
| X | IM | 15 | 15 | 18 | | | | |



EXAMPLE:

If register R5 contains %0400, the byte at location %0400 contains 2, and the source operand is the immediate value 3, the statement

CPB @R5,#3

will leave the C flag set, indicating a borrow, the S flag set, and the Z and V flags cleared.

Compare and Decrement

CPD CPDB dst, src, r, cc

dst: R src: IR

INSTRUCTION FORMAT: (F6.5)

src 1 1 0 0 0 0

dst src

OPERATION:

dst - src AUTODECREMENT src {-1 if byte, -2 if word} r < -r - 1

This instruction is used to search a string of data for an element meeting the specified condition. The contents of the location addressed by the source register is compared to (subtracted from) the destination operand, and the Z flag is set if the condition code specified by "cc" would be set by the comparison; otherwise the Z flag is cleared. See section 3.2.1 for a list of condition codes. Both operands are unaffected.

The source register is then decremented by one if CPDB, or by two if CPD, thus moving the pointer to the previous element in the string. The word register specified by "r" (used as a counter) is then decremented by one.

- FLAGS: C: Undefined
 - Z: Set if the condition code generated by the comparison matches cc; cleared otherwise
 - S: Undefined
 - V: Set if the result of decrementing r is zero; cleared otherwise
 - D: Unaffected
 - H: Unaffected

CYCLES:

Word/Byte 20

EXAMPLE:

If register RHO contains %FF, register Rl contains %4001, the byte at location %4001 contains %00, and register R3 contains 5, the statement

CPDB RHO, @Rl, R3, EQ

will leave the Z flag cleared since the condition code would not have been "equal". Register Rl will contain the value %4000 and R3 will contain 4.

Compare, Decrement and Repeat

CPDR dst,src,r,cc CPDRB dst: R src: IR

INSTRUCTION FORMAT: (F6.5)

 $\frac{\text{dst}}{R} \quad \frac{\text{src}}{IR} \quad \frac{r}{R}$

OPERATION:

dst - src
AUTODECREMENT src {-1 if byte, -2 if word}
r <- r - 1</pre>

repeat until cc is true or r = 0

This instruction is used to search a string of data for an element meeting the specified condition. The contents of the location addressed by the source register is compared to (subtracted from) the destination operand, and the Z flag is set if the condition code specified by "cc" would be set by the comparison; otherwise the Z flag is cleared. See section 3.2.1 for a list of condition codes. Both operands are unaffected.

The source register is then decremented by one if CPDRB, or by two if CPDR, thus moving the pointer to the previous element in the string. The word register specified by "r" (used as a counter) is then decremented by one. The entire operation is repeated until either the condition is met or the result of decrementing r is zero. This instruction can search a string from 1 to 65536 bytes or 32768 words long (the value of r must not be greater than 32768 for CPDR).

FLAGS: C: Undefined

Z: Set if the condition code generated by the comparison matches cc; cleared otherwise

S: Undefined

V: Set if the result of decrementing r is zero; cleared otherwise

D: Unaffected
H: Unaffected

CYCLES: Wor

Word/Byte 11+9*n (n=number of data elements compared)

Compare, Decrement and Repeat

NOTE:

This instruction can be interrupted after each execution of the basic operation. The program counter value of the start of this instruction is saved before the interrupt request is accepted, so that the instruction can be properly resumed. Seven more cycles should be added to this instruction's execution time for each interrupt request that is accepted.

EXAMPLE:

If the string of words starting at location %2000 contain the values 0,2,4,6 and 8, register R2 contains %2008, R3 contains 3, and R8 contains 8, the statement

CPDR R3, @R2, R8, GT

will leave the Z flag set indicating the condition was met. Register R2 will contain the value %2000, R3 will still contain 3, and R8 will contain 4.

Compare and Increment

CPI dst,src,r,cc

dst: R src: IR

INSTRUCTION FORMAT: (F6.5)

W/B: 1 0 1 1 1 0 1 W/B src 0 0 0

 $\frac{\text{dst}}{R} \quad \frac{\text{src}}{IR} \quad \frac{r}{R}$

OPERATION:

0 0 0 0

dst - src
AUTOINCREMENT src {+1 if byte, +2 if word}
r <- r - 1</pre>

CC

This instruction is used to search a string of data for an element meeting the specified condition. The contents of the location addressed by the source register is compared to (subtracted from) the destination operand, and the Z flag is set if the condition code specified by "cc" would be set by the comparison; otherwise the Z flag is cleared. See section 3.2.1 for a list of condition codes. Both operands are unaffected.

The source register is then incremented by one if CPIB, or by two if CPI, thus moving the pointer to the next element in the string. The word register specified by "r" (used as a counter) is then decremented by one.

FLAGS:

- C: Undefined
- Z: Set if the condition code generated by the comparison matches cc; cleared otherwise
- S: Undefined
- V: Set if the result of decrementing r is zero; cleared otherwise
- D: Unaffected
- H: Unaffected

CYCLES:

Word/Byte

EXAMPLE:

This instruction is used in a "loop" of instructions which searches a string of data for an element meeting the specified condition, but an intermediate operation on each data element is required. The following sequence "scans while numeric", that is, a string is searched until either an ASCII character not in the

Compare and Increment

range "0" to "9" (see Appendix) is found, or the end of the string is reached. This involves a range check on each character (byte) in the string.

| | LD LD | R3,#STRLEN R1,#STRSTART RL0,#'9' | !Initialize counter! !Load start address! !Largest numeric char! |
|-------|------------|--|--|
| LOOP: | CPB | @R1,#'0' | !Test char < '0'! |
| | JR CPIB | ULT, NONNUMERIC RLO, @Rl, R3, ULT | !Test char > '9'! |
| | JR JR | Z, NONNUMERIC NOV, LOOP | !Repeat until counter=0! |
| DONE: | | | |

tents of the location addressed by the source register

NONNUMERIC: !Handle non-numeric char!

CPIR dst,src,r,cc

dst: R src: IR

INSTRUCTION FORMAT: (F6.5)

 $\frac{\text{dst}}{R} \quad \frac{\text{src}}{IR} \quad \frac{r}{R}$

OPERATION:

dst - src
AUTOINCREMENT src {+1 if byte, +2 if word}
r <- r - 1
repeat until cc is true or r = 0</pre>

This instruction is used to search a string of data for an element meeting the specified condition. The contents of the location addressed by the source register is compared to (subtracted from) the destination operand, and the Z flag is set if the condition code specified by "cc" would be set by the comparison; otherwise the Z flag is cleared. See section 3.2.1 for a list of condition codes. Both operands are unaffected.

The source register is then incremented by one if CPIRB, or by two if CPIR, thus moving the pointer to the next element in the string. The word register specified by "r" (used as a counter) is then decremented by one. The entire operation is repeated until either the condition is met or the result of decrementing r is zero. This instruction can search a string from 1 to 65536 bytes or 32768 words long (the value of r must not be greater than 32768 for CPIR).

FLAGS: C: Undefined

Z: Set if the condition code generated by the comparison matches cc; cleared otherwise

S: Undefined

V: Set if the result of decrementing r is zero; cleared otherwise

D: Unaffected
H: Unaffected

CYCLES: Word/Byte

11+9*n (n=number of data elements compared)

NOTE: This instruction can be interrupted after each execution of the basic operation. The program counter value of the start of this instruction is saved

before the interrupt request is accepted, so that the instruction can be properly resumed. Seven more cycles should be added to this instruction's execution time for each interrupt request that is accepted.

EXAMPLE:

The following sequence of instructions can be used to search a string for an ASCII return character. The pointer to the start of the string is set, the string length is set, the character (byte) to be searched for is set, and then the search is accomplished. Testing the Z flag determines whether the character was found.

LD R1,#STRSTART
LD R3,#STRLEN LD RLO, #'%R' CPIRB RLO, @Rl, R3, EQ JR Z, FOUND set if the condition code specifies by were would be as a beginning the companies to a line of the condition codes. But here declines a list of condition codes.

Compare String and Decrement

CPSD dst,src,r,cc med dst: IR CPSDB

vinegoig ed src: IR

INSTRUCTION FORMAT: (F6.5)

0 0 CC

dst src r $\frac{\text{dst}}{\text{1}} \frac{\text{src}}{\text{IR}} \frac{\text{r}}{\text{R}}$

OPERATION: dst - src

AUTODECREMENT dst and src {-1 if byte, -2 if word} onuol saw isir <- r - 1

This instruction is used to compare two strings of data until the specified condition is true. The contents of the location addressed by the source register is compared to (subtracted from) the contents of the location addressed by the destination register. The Z flag is set if the condition code specified by "cc" would be set by the comparison; otherwise the Z flag is cleared. See section 3.2.1 for a list of condition codes. operands are unaffected.

The source and destination registers are then decremented by one if CPSDB, or by two if CPSD, thus moving the pointers to the previous elements in the strings. word register specified by "r" (used as a counter) is then decremented by one.

FLAGS:

C: Undefined

Z: Set if the condition code generated by the comparison matches cc; cleared otherwise

S: Undefined

V: Set if the result of decrementing r is zero; cleared

otherwise D: Unaffected H: Unaffected

CYCLES:

Word/Byte 25

EXAMPLE:

If register R2 contains %2000, the byte at location %2000 contains %FF, register R3 contains %3000, the byte at location %3000 contains %00, and register R4 contains 1, the statement

Compare String and Decrement

CPSDB @R2, @R3, R4, UGE

will leave the Z flag set since the condition code would have been "unsigned greater than or equal", and the V flag will be set to indicate that the counter R4 now contains 0. R2 will contain %1FFF, and R3 will contain %2FFF.

Compare String, Decrement and Repeat

CPSDR dst,src,r,cc dst dst: IR CPSDRB src: IR

INSTRUCTION FORMAT: (F6.5)

W/B: 1 0 1 1 0 1 W/B src 1 1 1 0 TR TR TR

OPERATION:

dst - src AUTODECREMENT dst and src $\{-1 \text{ if byte, } -2 \text{ if word}\}$ r $\langle -r-1 \rangle$ repeat until cc is true or r = 0

This instruction is used to compare two strings of data until the specified condition is true. The contents of the location addressed by the source register is compared to (subtracted from) the contents of the location addressed by the destination register. The Z flag is set if the condition code specified by "cc" would be set by the comparison; otherwise the Z flag is cleared. See section 3.2.1 for a list of condition codes. Both operands are unaffected.

The source and destination registers are then decremented by one if CPSDRB, or by two if CPSDR, thus moving the pointers to the previous elements in the strings. The word register specified by "r" (used as a counter) is then decremented by one. The entire operation is repeated until either the condition is met or the result of decrementing r is zero. This instruction can compare strings from 1 to 65536 bytes or 32768 words long (the value of r must not be greater than 32768 for CPSDR).

FLAGS: C: Undefined

Z: Set if the condition code generated by the comparison matches cc; cleared otherwise

S: Undefined

V: Set if the result of decrementing r is zero; cleared otherwise

D: Unaffected
H: Unaffected

CYCLES:

 $\frac{\text{Word/Byte}}{11+14*n} \text{ (n=number of data elements compared)}$

Compare String, Decrement and Repeat

NOTE:

This instruction can be interrupted after each execution of the basic operation. The program counter of the start of this instruction is saved before the interrupt request is accepted, so that the instruction can be properly resumed. Seven more cycles should be added to this instruction's execution time for each interrupt request that is accepted.

EXAMPLE:

If the words from location %1000 to %1006 contain the values 0,2,4, and 6, the words from location %2000 to %2006 contain the values 0,1,1,0, register R13 contains %1006, register R14 contains %2006, and register RO contains 4, the statement

CPSDR @R13,@R14,R0,EQ

leaves the Z flag set since the condition code would have been "equal" (locations %1000 and %2000 both contain the value 0). R13 will contain %0FFE, R14 will contain %1FFE, and RO will contain O.

Compare String and Increment

CPSI dst,src,r,cc

dst: IR src: IR

INSTRUCTION FORMAT: (F6.5)

 $\frac{\text{dst}}{\text{IR}}$ $\frac{\text{src}}{\text{IR}}$ $\frac{\text{r}}{\text{R}}$

OPERATION:

dst - src
AUTOINCREMENT dst and src {+1 if byte, +2 if word}
r <- r - 1</pre>

This instruction is used to compare two strings of data until the specified condition is true. The contents of the location addressed by the source register is compared to (subtracted from) the contents of the location addressed by the destination register. The Z flag is set if the condition code specified by "cc" would be set by the comparison; otherwise the Z flag is cleared. See section 3.2.1 for a list of condition codes. Both operands are unaffected.

The source and destination registers are then incremented by one if CPSIB, or by two if CPSI, thus moving the pointers to the next elements in the strings. The word register specified by "r" (used as a counter) is then decremented by one.

FLAGS:

- C: Undefined
- Z: Set if the condition code generated by the comparison matches cc; cleared otherwise
- S: Undefined
- V: Set if the result of decrementing r is zero; cleared otherwise
- D: Unaffected
- H: Unaffected

CYCLES:

Word/Byte

EXAMPLE:

This instruction is used in a "loop" of instructions which compares two strings until the specified condition is true, but where an intermediate operation on each data element is required. The following sequence attempts to match a given source string to the destination string which is known to contain all upper-case characters. The match should succeed even if the source

Compare String and Increment

string contains some lower-case characters. This involves a forced conversion of the source string to upper-case (only ASCII alphabetic letters are assumed, see Appendix) by resetting bit 5 of each character (byte) before comparison.

| | LD | R1,#SRCSTART | !Load start addresses! |
|-------|---------------------------|---|---|
| LOOP: | LD LD | R2, #DSTSTART R3, #STRLEN | !Initialize counter! |
| DONE: | RESB CPSIB JR JR | @R1,#5 @R1,@R2,R3,NE Z,NOTEQUAL NOV,LOOP | !Force upper-case! !Compare until not equal! !Exit loop if match fails! !Repeat until counter=0! !Match succeeds! |
| | | | |

NOTEQUAL: 932 por norden deeb !Match fails! Dos ed bluw 90 vd beilipega eboo got bened ed bluw 90 vd beilipega eboo got bened ed bluw 90 vd dee

cared to (subtracted from) the contents of the

Compare String, Increment and Repeat

CPSIR dst,src,r,cc dst: IR CPSIRB

src: IR

INSTRUCTION FORMAT: (F6.5)

0 0 0 0

dst TR

OPERATION:

dst - src

AUTOINCREMENT dst and src {+1 if byte, +2 if word}

r < -r - 1

repeat until cc is true or r = 0

This instruction is used to compare two strings of data until the specified condition is true. The contents of the location addressed by the source register is compared to (subtracted from) the contents of the location addressed by the destination register. The Z flag is set if the condition code specified by "cc" would be set by the comparison; otherwise the Z flag is cleared. See section 3.2.1 for a list of condition codes. Both operands are unaffected.

The source and destination registers are then incremented by one if CPSIRB, or by two if CPSIR, thus moving the pointers to the next elements in the strings. The word register specified by "r" (used as a counter) is then decremented by one. The entire operation is repeated until either the condition is met or the result of decrementing r is zero. This instruction can compare strings from 1 to 65536 bytes or 32768 words long (the value of r must not be greater than 32768 for CPSIR).

FLAGS:

- C: Undefined
- Z: Set if the condition code generated by the comparison matches cc; cleared otherwise
- S: Undefined
- V: Set if the result of decrementing r is zero; cleared otherwise
- D: Unaffected
- H: Unaffected

CYCLES:

Word/Byte

11+14*n (n=number of data elements compared)

Compare String, Increment and Repeat

NOTE:

This instruction can be interrupted after each execution of the basic operation. The program counter of the start of this instruction is saved before the interrupt request is accepted, so that the instruction can be properly resumed. Seven more cycles should be added to this instruction's execution time for each interrupt request that is accepted.

EXAMPLE:

The following sequence of instructions can be used to compare two strings of possibly different lengths, but at least one character (byte) long. It determines if the destination string is lexicographically greater than or equal to the source string as determined by the ordering of the ASCII character set (see Appendix). Notice that the string "ABC" is greater than "AB", and "AC" is greater than "ABC". The destination string is initially pointed to by Rl, with its length in bytes in R3. The source string is initially pointed to by R2,

| | with i | ts lengt | h in bytes | in R4. | |
|---|--------|----------|--------------|--------|-------------------------|
| | | CP 0-0 | R3,R4 | | !Find shortest length! |
| | | JR | ULE, COMPAI | RE 8-0 | |
| | | LD 8-0 | R3,R4 | | !If source is shorter,! |
| | | EX | R1,R2 | | !then swap pointers! |
| | COMPAR | RE: 9-A | 0 | | 0 |
| | | CPSIRB | @R1, @R2, R3 | 3,ULT | !Compare while >=! |
| | | JR 9-0 | Z,FAIL | | DE RIDIE |
| | SUCCE | ED: T-A | . 0 | | !Destination >= source! |
| | | | . 0 | | |
| | FAIL: | | | | !Destination < source! |
| | OA | 6-0 | | | |
| I | | | | 4-3 | |

Decimal Adjust

DAB dst dst: R

INSTRUCTION FORMAT: (F1.1) on mayor complete to

0 0 0 0 10 R B: mode 1 1 0 0 0 0

mode dst

OPERATION: dst <- DA dst

The destination byte is adjusted to form two 4-bit BCD digits following an addition or subtraction operation. For addition (ADDB, ADCB) or subtraction (SUBB, SBCB), the following table indicates the operation performed:

| Instruction | Carry Before DAB | Bits 4-7 Value (Hex) | H Flag Before DAB | Bits 0-3 Value (Hex) | Number Added To Byte | Carry After DAB |
|--------------|--------------------------------------|--|---------------------------------|--|--|---------------------------------|
| ADDB ADCB | 0 0 0 0 0 0 0 1 | 0-9 0-8 0-9 A-F 9-F A-F 0-2 0-2 | 0 0 1 0 0 1 0 | 0-9 A-F 0-3 0-9 A-F 0-3 0-9 A-F | 00 06 06 60 66 66 60 | 0 0 0 1 1 1 1 |
| SUBB SBCB | 0 0 1 1 | 0-9 0-8 7-F 6-F | 0 1 0 1 | 0-9 6-F 0-9 6-F | 00 FA A0 9A | 0 0 1 1 |

The operation is undefined if the destination byte was not the result of a valid addition or subtraction of BCD digits.

FLAGS:

C: Set or cleared according to the table above

Z: Set if the result is zero; cleared otherwise

S: Set if the most significant bit of the result is set;

cleared otherwise V: Unaffected

D: Unaffected H: Unaffected

CYCLES:

EXAMPLE:

If addition is performed using the BCD values 15 and 27, the result should be 42. The sum is incorrect, however, when the binary representations are added in the destination location using standard binary arithmetic.

 $\begin{array}{cccc}
0001 & 0101 \\
+ & 0010 & 0111 \\
\hline
0011 & 1100 & = %3C
\end{array}$

The DAB instruction adjusts this result so that the correct BCD representation is obtained.

 $\begin{array}{c} + \begin{array}{c} 0011 & 1100 \\ 0000 & 0110 \\ \hline 0100 & 0010 \end{array} = 42 \end{array}$

DEC Decrement

DEC dst,src

dst: R, IR, DA, X src: IM

dst

INSTRUCTION FORMAT: (F1.2)

W/B: mode 1 0 1 0 1 W/B dst src

10 R IM 00 IR IM 01 DA IM (dst field=0) 01 X IM (dst field<>0)

STC

OPERATION: dst <- dst - src {src is 1 to 16}

The source operand (a value from 1 to 16) is subtracted from the destination operand and the result is stored in the destination. Subtraction is performed by adding the two's complement of the source operand to the destination operand. The source operand may be omitted from the assembly language statement and thus defaults to the value 1.

mode

FLAGS: C: Unaffected

Set if the result is zero; cleared otherwiseSet if the result is negative; cleared otherwiseV: Set if arithmetic overflow occurs, that is, the sign of the destination was opposite the sign of the result: cleared otherwise

D: Unaffected H: Unaffected

CYCLES: dst Word/Byte NS SS SL R 4 -- --IR 11 -- --DA 13 14 16

X

NOTE: The "src field" in the instruction format encoding contains the source operand. The "src field" values range from 0 to 15 corresponding to the source values 1 to 16.

EXAMPLE: If register R10 contains %002A, the statement

DEC R10

14

14 17

will leave the value %0029 in R10.

DI int

int: VI, NVI

INSTRUCTION FORMAT: (F9.2)

0 1 1 1 1 1 0 0 0 0 0 0 0 0 VI NVI

OPERATION: if instruction(0) = 0 then NVI <- 0 if instruction(1) = 0 then VI <- 0

Any combination of the Vectored Interrupt (VI) or Non-Vectored Interrupt (NVI) control bits in the Flags and Control Word (FCW) are cleared to zero if the corresponding bit in the instruction is zero, thus disabling the appropriate type of interrupt. If the corresponding bit in the instruction is one, the control bit will not be affected. All other bits in the FCW are not affected. There may be one or two operands in the assembly language statement, in either order.

FLAGS: No flags affected without a tyrisby sydeb

CYCLES: d 507 (vib at (5nebivib) bnarego notionidaeb ed

NOTE: This is a privileged instruction.

EXAMPLE: If both the NVI and VI control bits are set (=1), the statement

same sign as the dividend. For DIV ID destination is

will leave the NVI control bit set (=1), and the VI control bit cleared (=0).

a register pair and the source is a word

DIV dst.src DTVI

dst · R

src: R, IM, IR, DA, X

INSTRUCTION FORMAT: (F2.1)

| mode | 0 | 1 | 1 | 0 | 1 | 1 | src | dst |
|------|---|---|---|---|---|---|-----|-----|
| mode | 0 | 1 | 1 | 0 | 1 | 0 | src | dst |

 $\frac{\text{mode}}{10} \frac{\text{dst}}{R} \frac{\text{src}}{R}$ 00 R IM (src field=0) 00 R IR (src field<>0) 01 R DA (src field=0) 01 R $\times (\text{src field} <> 0)$

OPERATION: Word

dst,dstyl <- dst,dstyl / src {dst <- remainder nos and particles | dstyl <- quotient}

dst,dstv1,dstv2,dstv3 <- dst,dstv1,dstv2,dstv3 /

{dst.dstv/l <- remainder dstv2,dstv3 <- quotient}

The destination operand (dividend) is divided by the source operand (divisor), the quotient is stored in the low-order half of the destination and the remainder is stored in the high-order half of the destination. The contents of the source are not affected. Both operands are treated as signed, two's complement integers and division is performed so that the remainder is of the same sign as the dividend. For DIV, the destination is a register pair and the source is a word value; for DIVL, the destination is a register quadruple and the source is a long word value.

FLAGS:

- C: DIV set if quotient is less than -2^{15} or not less than 215; cleared otherwise; DIVL - set if quotient is less than -2^{31} or not less than 2^{31} ; cleared otherwise
- Z: Set if the quotient or divisor is zero; cleared otherwise
- S: Set if the quotient is negative; cleared otherwise (contains the sign of the divisor if the instruction is aborted)
- V: Set if the divisor is zero or if the quotient would be too large to fit in the low-order half of the destination operand (the instruction is aborted if the absolute value of the high-order half of the dividend is larger than the absolute value of the divisor); cleared otherwise
- D: Unaffected
- H: Unaffected

| CYCLES: | src | | Word | | | Long | | |
|---------|-------|---------------|----------|--------|--------------------|---------------|------------------|----------------|
| | | NS | SS | SL | NS | SS | SL | |
| | R | 95 | | | 723 | | | |
| | IM | 95 | | | 723 | F3-1) | TA | |
| | IR | 95 | | | 723 | | | |
| | DA | 96 | 97 | 99 | 724 | 725 | 727 | |
| | X | 97 | 97 | 100 | 725 | 725 | 728 | |
| | | | | | | | | |
| | (Divi | isor is | s zer | 0) | | | | |
| | R | 13 | TU GO UT | 5 == | 30 | ind-1 | egiete | |
| | IM | 13 | 9== | istali | 30 | 38 | ontent | |
| | IR | 13 | 11 | 88-46 | 30 | 5.n ± ± 8 | the-de | |
| | DA | 14 | 15 | 17 | 31 | 32 | 34 | |
| | X | 15 | 15 | 18 | 32 | 32 | 35 | |
| | | | | | | | | |
| | divid | olute dend is | s lar | of the | he high han the | -orde abso | r half lute v | of the alue of |
| | | | | | | | | |

NOTE:

For proper instruction execution, the "dst field" in the instruction format encoding must be even for DIV, and must be a multiple of 4 (0,4,8,12) for DIVL. If the source operand in DIVL is a register, the "src field" must be even.

51

53

53

55

56

51

51

52

53

EXAMPLE:

If register RRO (composed of word registers RO and R1) contains %00000022 and register R3 contains 6, the statement

ment by subtracting the PC RR, ORR VICE tollowing

25

25

25

26

27

27

27

28

29

R

IM

IR

DA

X

will leave the value %00040005 in RRO (Rl contains the quotient 5 and RO contains the remainder 4).

Decrement and Jump if Not Zero

DJNZ r,dst DBJNZ

dst: RA

dst

INSTRUCTION FORMAT: (F3.1)

W/B: 1 1 1 WIR

OPERATION:

r <- r - 1 if r <> 0 then PC <- PC - (2*disp)

The register being used as a counter is decremented. If the contents of the register are not zero after decrementing, the destination address is calculated and then loaded into the program counter (PC). Control will then pass to the instruction whose address is pointed to by the PC. When the register counter reaches zero, control falls through to the instruction following DJNZ or DBJNZ. This instruction provides a simple, efficient method of loop control.

FLAGS:

No flags affected

CYCLES:

Word/Byte

The relative addressing mode is calculated by doubling the displacement in the instruction, then subtracting this value from the updated value of the PC to derive the destination address. The updated PC value is taken to be the address of the instruction byte following the DJNZ or DBJNZ instruction, while the displacement is a 7-bit positive value in the range 0 to 127. Thus, the destination address must be in the range -252 to 2 bytes from the start of the DJNZ or DBJNZ instruction. In the segmented mode, the PC segment number is not affected. The assembler automatically calculates the displacement by subtracting the PC value of the following instruction from the address given by the programmer. Note that DJNZ or DBJNZ can not be used to transfer control in the forward direction.

EXAMPLE:

DJNZ and DBJNZ are typically used to control a "loop" of instructions. In this example, 100 bytes are moved from one buffer area to another, and the sign bit of

each byte is cleared to zero. Register RHO is used as the counter.

| | LDB LD LD | RH0,#100 R1,#SRCBUF R2,#DSTBUF | !Initialize counter! !Load start addresses! |
|--------------|---------------------------|---------------------------------------|---|
| LOOP: | 1 -> 1/ | 0) = 0 then N | |
| rupt oits | LDB RESB LDB INC | RLO, @R1 RLO, #7 @R2, RLO R1 | !Load source byte! !Mask off sign bit! !Store into destination! !Advance pointers! |
| | INC DBJNZ | R2 RH0,LOOP | !Repeat until counter=0! |

enabling the appropriate type of int:TX3N . If the corresponding bit in the instruction is one, the control bit will not be affected. All other bits

127

EI int int: VI, NVI

INSTRUCTION FORMAT: (F9.2)

0 1 1 1 1 1 0 0 0 0 0 0 0 1 VI NVI

OPERATION:

if instruction(0) = 0 then NVI <- 1 if instruction(1) = 0 then VI <- 1

Any combination of the Vectored Interrupt (VI) or Non-Vectored Interrupt (NVI) control bits in the Flags and Control Word (FCW) are set to one if the corresponding bit in the instruction is zero, thus enabling the appropriate type of interrupt. If the corresponding bit in the instruction is one, the control bit will not be affected. All other bits in the FCW are not affected. There may be one or two operands in the assembly language statement, in either order.

FLAGS:

No flags affected

CYCLES:

7

NOTE:

This is a privileged instruction.

EXAMPLE:

If the NVI control bit is set (=1), and the VI control bit is clear (=0), the statement

EI NVI, VI

will leave both the NVI and VI control bits set (=1).

dst,src EX EXB

dst: R src: R, IR, DA, X

INSTRUCTION FORMAT: (F2.1)

1 0 1 1 0 W/B W/B: mode

00 R IR 01 R DA (src field=0) R X (src field<>0) 01

mode dst src R

OPERATION:

tmp <- src

{tmp is an internal register}

src <- dst dst <- tmp

The contents of the source operand are exchanged with the contents of the destination operand.

FLAGS:

No flags affected and about 0 = (21) 10 ab 31

CYCLES:

| src | Wo | rd/B | yte |
|-----|----|-------|-------|
| | NS | SS | SL |
| R | 6 | R75a | 5 tra |
| IR | 12 | Iv-za | Була |
| DA | 15 | 16 | 18 |
| X | 16 | 16 | 19 |

The sign bit of the low-ordeel all of the operand is copied into all bit positions EXAMPLE: If register RO contains 8 and register R5 contains 9, the statement of add and and and and by

entine notate EX = RO, R5 m of fulegy at notionstant wide

will leave the values 9 in RO, and 8 in R5.

129

EXTS Extend Sign

EXTS dst

dst: R

EXTSB EXTSL

INSTRUCTION FORMAT: (F1.1)

| | | | 38 | | 724 | | | 0.1 | | | | 180 | mode | dst |
|----|------|---|----|---|-----|---|---|-----|----|---|---|-----|------|-----|
| B: | mode | 1 | 1 | 0 | 0 | 0 | 1 | dst | 0 | 0 | 0 | 0 | 10 | R |
| W: | mode | 1 | 1 | 0 | 0 | 0 | 1 | dst | 1 | 0 | 1 | 0 | | |
| L: | mode | 1 | 1 | 0 | 0 | 0 | 1 | dst | 10 | 1 | 1 | 1 | | |

OPERATION: Byte

if dst(7)=0 then dst(8:15) <- 000...000 else dst(8:15) <- 111...111

 $\frac{\text{Word}}{\text{if dst}_{V}1(15)=0}$ then $\text{dst}(16:31) \leftarrow 000...000$ else $\text{dst}(16:31) \leftarrow 111...111$

Long if $dst_{V}^{2}, dst_{V}^{3}(31) = 0$ then $dst_{V}^{2}(32:63) < -000...000$ else $dst_{V}^{3}(32:63) < -111...111$

The sign bit of the low-order half of the destination operand is copied into all bit positions of the high-order half of the destination. For EXTSB, the destination is a word register; for EXTS, the destination is a register pair; for EXTSL, the destination is a register quadruple. This instruction is useful in multiple precision arithmetic or for conversion of small signed operands to larger signed operands.

FLAGS:

No flags affected

CYCLES:

Word/Byte

Long

NOTE:

For proper instruction execution, the "dst field" in the instruction format encoding must be even for EXTS, and must be a multiple of $4\ (0,4,8,12)$ for EXTSL.

EXAMPLE:

If register pair RR2 (composed of word registers R2 and R3) contains %12345678, the statement

EXTS RR2

will leave the value \$00005678 in RR2 (because the sign bit of R3 was 0).

HALT

INSTRUCTION FORMAT: (F9.3)

0 1 1 1 1 0 1 0 0 0 0 0 0 0 0 0

OPERATION:

The CPU operation is suspended until an interrupt or reset request is received. This instruction is used to synchronize the Z8000 with external events, preserving its state until an interrupt or reset request is honored. After an interrupt is serviced, control falls through to the instruction

following the HALT.

FLAGS: No flags affected

CYCLES: 8+3*n (Interrupts are recognized at the end of

each 3-cycle period; thus n=number of

periods without interruption)

NOTES: This is a privileged instruction.

While in the halt state, memory refresh cycles will

still occur, and BUSRQ will be honored.

IN dst, src dst: R src: IR, DA INB INSTRUCTION FORMAT: (F7.3)dst src TR dst W/B: 0 0 1 1 1 1 0 W/B src (F7.1)dst src R DA W/B: 0 0 1 1 1 0 1 W/B dst

OPERATION:

dst <- src

The contents of the source operand are loaded into the destination. I/O addresses are always 16 bits.

FLAGS: No flags affected

CYCLES: $\frac{\text{src}}{\text{IR}}$ $\frac{\text{Word/Byte}}{10}$

DA 12

NOTE: This is a privileged instruction.

EXAMPLE: If register R6 contains the I/O address %0123, and the

"port" %0123 contains %FF, the statement

INB RH2, @R6

will leave the value %FF in register RH2.

TNC dst, src TNCB

dst: R, IR, DA, X

src: IM

INSTRUCTION FORMAT: (F1.2)

| | | | | | | | | | | | mode | dst | src | | |
|-------|------|---|-----|---|---|-----|---|-----|---|-----|------|-----|-----|------|-----------|
| W/B:[| mode | 1 | 0 1 | 0 | 0 | W/B | | dst | | sro | 10 | R | IM | | |
| ", " | | | 1 | - | | _ | 1 | | 1 | 11 | 00 | IR | IM | | |
| | | | | | | | | | | | 01 | DA | IM | (dst | field=0) |
| | | | | | | | | | | | 01 | X | IM | (dst | field<>0) |

OPERATION: dst <- dst + src {src is 1 to 16}

The source operand (a value from 1 to 16) is added to the destination operand and the sum is stored in the destination. Two's complement addition is performed. The source operand may be omitted from the assembly language statement and thus defaults to the value 1.

FLAGS: C: Unaffected

Z: Set if the result is zero; cleared otherwise

S: Set if the result is negative; cleared otherwise

V: Set if arithmetic overflow occurs, that is, the sign of the destination was opposite the sign of the

result; cleared otherwise

D: Unaffected H: Unaffected

Word/Byte CYCLES: dst NS SS SL R 1 11 TR 13 14 16 DA 14 14 17 X

The "src field" in the instruction format encoding contains NOTE: the source operand. The "src field" values range from 0 to 15 corresponding to the source values 1 to 16.

If register RH2 contains %21, the statement EXAMPLE:

INCB RH2, #6

will leave the value %27 in RH2.

IND

Input and Decrement

IND dst,src,r

dst: IR src: IR

INSTRUCTION FORMAT: (F6.4)

W/B: 0

| 0 0 | 1 | 1 | 1 | 0 | 1 | W/B | src | 1 | 0 | 0 | 0 |
|-----|---|---|---|---|---|------|-----|---|---|---|---|
| 0 0 | 0 | 0 | | | - | 1 10 | dst | 1 | 0 | 0 | 0 |

 $\frac{\text{dst}}{\text{IR}}$ $\frac{\text{src}}{\text{IR}}$ $\frac{\text{r}}{\text{R}}$

OPERATION:

dst <- src

AUTODECREMENT dst $\{-1 \text{ if byte, } -2 \text{ if word}\}$ r $\langle -r-1$

This instruction is used for block input of strings of data. The contents of the I/O location addressed by the source register are loaded into the memory location addressed by the destination register. I/O addresses are always 16 bits. The destination register is then decremented by one if INDB, or by two if IND, thus moving the pointer to the previous element in the string. The word register specified by "r" (used as a counter) is then decremented by one.

FLAGS:

- C: Unaffected
- Z: Undefined
- S: Unaffected
- V: Set if the result of decrementing r is zero; cleared
 - otherwise
- D: Unaffected
- H: Unaffected

CYCLES:

Word/Byte 21

NOTE:

This is a privileged instruction.

EXAMPLE:

In segmented mode, if register RR4 contains \$02004000 (segment 2, offset \$4000), register R6 contains the I/O address \$0228, the "port" \$0228 contains \$05B9, and register R0 contains \$0016, the statement

IND @RR4, @R6, RO

will leave the value \$05B9 in location \$02004000, the value \$02003FFE in RR4, and the value \$0015 in R0. Register R6 still contains the value \$0228.

Input, Decrement and Repeat

INDR dst,src,r dst dst: IR INDRB

src: IR

INSTRUCTION FORMAT: (F6.4)

| W/B: | 0 | 0 | 1 | , 1 | 1 | 0 | 1 | W/B | src | 1 | 0 | 0 | 0 |
|------|---|---|---|-----|-----|---|---|-----|-----|---|---|---|---|
| пі | 0 | 0 | 0 | 0 | 9.5 | | r | 5 | dst | 0 | 0 | 0 | 0 |

dst src r IR IR

OPERATION:

dst <- src

AUTODECREMENT dst {-1 if byte, -2 if word}

r <- r - 1

repeat until r = 0

This instruction is used for block input of strings of data. The contents of the I/O location addressed by the source register are loaded into the memory location addressed by the destination register. I/O addresses are always 16 bits. The destination register is then decremented by one if INDRB, or by two if INDR, thus moving the pointer to the previous element in the string. The word register specified by "r" (used as a counter) is then decremented by one. The entire operation is repeated until the result of decrementing r is zero. This instruction can input from 1 to 65536 bytes or 32768 words (the value for r must not be greater than 32768 for INDR).

FLAGS: C: Unaffected

7: Undefined

S: Unaffected

V: Set

D: Unaffected

H: Unaffected

CYCLES:

Word/Byte

11+10*n (n=number of data elements transferred)

NOTES:

This is a privileged instruction.

This instruction can be interrupted after each execution of the basic operation. The program counter value of the start of this instruction is saved before the interrupt request is accepted, so that the instruction can be properly resumed. Seven more cycles should be added to this instruction's execution time for each interrupt request that is accepted.

INDR

Input, Decrement and Repeat

EXAMPLE:

If register R1 contains \$202A, register R2 contains the I/O address \$0050, and register R3 contains 8, the statement

INDRB @R1, @R2, R3

will input 8 bytes from "port" \$0050 and leave them in descending order from \$202A to \$2023. Register Rl will contain \$2022, and R3 will contain 0. R2 will not be affected.

136

INI dst,src,r

dst: IR src: IR

INSTRUCTION FORMAT: (F6.4)

dst src r IR IR R

OPERATION:

dst <- src
AUTOINCREMENT dst {+1 if byte, +2 if word}
r <- r - 1</pre>

This instruction is used for block input of strings of data. The contents of the I/O location addressed by the source register are loaded into the memory location addressed by the destination register. I/O addresses are always 16 bits. The destination register is then incremented by one if INIB, or by two if INI, thus moving the pointer to the next element in the string. The word register specified by "r" (used as a counter) is then decremented by one.

FLAGS: C: Unaffected

Z: Undefined

S: Unaffected

V: Set if the result of decrementing r is zero; cleared

otherwise
D: Unaffected
H: Unaffected

CYCLES: Word/H

Word/Byte

NOTE:

This is a privileged instruction.

EXAMPLE:

If register R4 contains \$4000, register Ro contains the I/O address \$0228, the "port" \$0228 contains \$B9, and register R0 contains \$0016, the statement

INIB @R4,@R6,R0

will leave the value %B9 in location %4000, the value %4001 in R4, and the value %0015 in R0. Register R6 still contains the value %0228.

Input, Increment and Repeat

TNTR dst.src.r TNTRB

dst: TR src: TR

INSTRUCTION FORMAT: (F6.4)

0 0 0 0 det 0 0 0 0 dst src TR

OPERATION:

dst <- src AUTOINCREMENT dst {+1 if byte, +2 if word} r <- r - 1 repeat until r = 0

This instruction is used for block input of strings of data. The contents of the I/O location addressed by the source register are loaded into the memory location addressed by the destination register. I/O addresses are always 16 bits. The destination register is then incremented by one if INIRB, or by two if INIR, thus moving the pointer to the next element in the string. The word register specified by "r" (used as a counter) is then decremented by one. The entire operation is repeated until the result of decrementing r is zero. This instruction can input from 1 to 65536 bytes or 32768 words (the value for r must not be greater than 32768 for INIR).

FLAGS: C: Unaffected

Z: Undefined S: Unaffected

V. Set

D: Unaffected

H: Unaffected

CYCLES:

Word/Byte

11+10*n (n=number of data elements transferred)

NOTES:

This is a privileged instruction.

This instruction can be interrupted after each execution of the basic operation. The program counter value of the start of this instruction is saved before the interrupt request is accepted, so that the instruction can be properly resumed. Seven more cycles should be added to this instruction's execution time for each interrupt request that is accepted.

EXAMPLE:

If register Rl contains \$2023, register R2 contains the I/O address \$0050, and register R3 contains 8, the statement

INIRB @R1, @R2, R3

will input 8 bytes from "port" %0050 and leave them in ascending order from %2023 to %202A. Register R1 will contain %202B, and R3 will contain 0. R2 will not be affected.

Interrupt Return

EXAMPLE: If register Ricontains #2023, register TERI

INSTRUCTION FORMAT: (F9.3)

0 1 1 1 1 0 1 1 0 0 0 0 0 0 0 0

OPERATION: Nonsegmented Segmented $SP \leftarrow SP + 2$ Segmented $SP \leftarrow SP + 2$ {Pop "identifier"} PS <- @SP PS <- @SP SP (- SP + 4 SP <- SP + 6

> This instruction is used to return to a previously executing procedure at the end of a procedure entered by an interrupt or trap (including a System Call instruction). First, the "identifier" word associated with the interrupt or trap is popped from the system processor stack and discarded. Then contents of the the location addressed by the system processor stack pointer are popped into the program status (PS), loading the Flags and Control Word (FCW) and the program counter (PC). The new value of the FCW is not effective until the next instruction, so that the status pins will not be affected by the new control bits until after the IRET instruction execution is completed. The next instruction executed is that addressed by the new contents of the PC. The system stack pointer (R15 if nonsegmented, or RR14 if segmented) is used.

C: Loaded from processor stack

Z: Loaded from processor stack S: Loaded from processor stack P/V: Loaded from processor stack D: Loaded from processor stack

H: Loaded from processor stack

CYCLES:

Address NS SS SL 13 -- 16

NOTES:

This is a privileged instruction.

The Z8001 version always executes the segmented mode of the IRET instruction.

IRET Interrupt Return

EXAMPLE:

In the nonsegmented Z8002 version, if the program counter contains %2550, the system stack pointer (R15) contains %3000, and locations %3000, %3002 and %3004 contain %7F03, a saved FCW value, and %1004, respectively, the statement

IRET

will leave the value %3006 in the system stack pointer and the program counter will contain %1004, the address of the next instruction to be executed. The program status is now determined by the saved FCW value.

JP cc,dst dst: IR, DA, X

INSTRUCTION FORMAT: (F1.3)

mode 0 1 1 1 1 0

mode dst TR 00 01 DA (dst field=0) 01 x (dst field<>0)

OPERATION: if cc is true then PC <- dst

A conditional jump transfers program control to the destination address if the condition specified by "cc" is true. See section 3.2.1 for a list of condition codes. If the condition code is true, the program counter (PC) is loaded with the designated address; otherwise,

control falls through to the instruction following the

JP instruction.

FLAGS:

No flags affected

CYCLES:

| | (jump | is t | aken) | (jump | not | taken) |
|-----------|-------|------|-------|-------|-------|--------|
| dst | Ad | dres | S | Ad | ddres | SS |
| trained . | NS | SS | SL | NS | SS | SL |
| IR | 10 | | 15 | 7 | | 7 |
| DA | 7 | 8 | 10 | 7 | 8 | 10 |
| X | 8 | 8 | 11 | 8 | 8 | 11 |

NOTE:

The destination address must be even since instructions

are word data.

EXAMPLE:

If the carry flag is set, the statement

JP C, %1520

replaces the contents of the program counter with %1520, thus transferring control to that location. JR cc,dst

dst: RA

INSTRUCTION FORMAT: (F3.3)

OPERATION:

if cc is true then PC <- PC + (2*disp)

A conditional jump transfers program control to the destination address if the condition specified by "cc" is true. See section 3.2.1 for a list of condition codes. If the condition code is true, the program counter (PC) is loaded with the designated address; otherwise, control falls through to the instruction following the JR instruction.

FLAGS:

No flags affected

CYCLES:

Address

6 (jump is taken) 6 (jump not taken)

NOTE:

The relative addressing mode is calculated by doubling the displacement in the instruction, then adding this value to the updated value of the PC to derive the destination address. The updated PC value is taken to be the address of the instruction byte following the JR instruction, while the displacement is a 8-bit signed value in the range -128 to +127. Thus, the destination address must be in the range -254 to +256 bytes from the start of the JR instruction. In the segmented mode, the PC segment number is not affected. The assembler automatically calculates the displacement by subtracting the PC value of the following instruction from the address given by the programmer.

EXAMPLES:

If the result of the last arithmetic operation executed is negative, the following four statements (which occupy a total of twelve bytes) are to be skipped. This can be accomplished with the statement

JR MI, \$+14

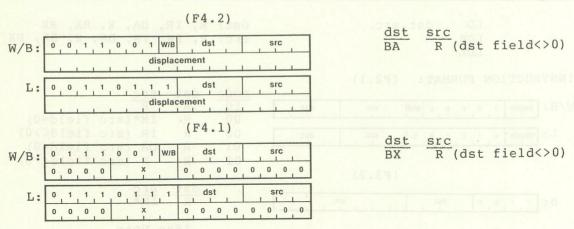
If the S flag is not set, execution continues with the statement following the JR.

A byte-saving form of a jump to the label LAB is

JR LAB

where LAB must be within the allowed range. The condition code is "blank" in this case, and indicates that the jump is always taken.

| LD dst,src LDB LDL | dst: R, IR, DA, X, BA, BX src: R, IM, IR, DA, X, BA, BX |
|--|--|
| INSTRUCTION FORMAT: (F2.1) W/B: mode 1 0 0 0 0 W/B src dst L: mode 0 1 0 1 0 0 src dst (F3.2) B: 1 1 0 0 dst src | mode dst src 10 R R 00 R IM*(src field=0) 00 R IR (src field<>0) 01 R DA (src field=0) 01 R X (src field<>0) |
| | *See Note |
| W/B: 0 0 1 1 0 0 0 W/B src dst displacement | $\frac{dst}{R} = \frac{src}{BA} $ (src field<>0) |
| L: 0 0 1 1 0 1 0 1 src dst displacement | |
| W/B: 0 1 1 1 0 0 0 W/B src dst 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 | dst src BX (src field<>0) |
| W/B: mode 1 0 1 1 1 W/B dst src L: mode 0 1 1 1 0 1 dst src | mode dst src 00 IR R 01 DA R (dst field=0) 01 X R (dst field<>0) |
| W/B: mode 0 0 1 1 0 W/B dst 0 1 0 1 W/B: add as a days wis reidment of the control of the contr | <pre>mode dst src 00 IR IM 01 DA IM (dst field=0) 01 X IM (dst field<>0)</pre> |



OPERATION: dst <- src

The contents of the source operand are loaded into the destination. The contents of the source are not affected.

| FLAGS: | No f | lags | affe | cted | | | | | | | | |
|-----------|------|------|----------|------|-----------|-------|------|-------|--|--|--|--|
| CYCLES: | dst | src | Wo NS | rd/B | yte SL | NS | Long | SL | | | | |
| | R | R | 3 | | | 5 | | FIRST | | | | |
| | R | IM | 7 | | | 11 | | | | | | |
| | R | IM | 5 | (by | te o | nly - | see | Note) | | | | |
| | R | IR | 7 | | | 11 | | | | | | |
| | R | DA | 9 | 10 | 12 | 12 | 13 | 15 | | | | |
| | R | X | 10 | 10 | 13 | 13 | 13 | 16 | | | | |
| | R | BA | 14 | | | 17 | | | | | | |
| | R | BX | 14 | | | 17 | | | | | | |
| | IR. | R | 8 | | | 11 | | 0 771 | | | | |
| | DA | R | 11 | 12 | 14 | 14 | 15 | 17 | | | | |
| | X | R | 12 | 12 | 15 | 15 | 15 | 18 | | | | |
| (0=1-1=13 | IR | IM | 11 | | | | | | | | | |
| | DA | IM | 14 | 15 | 17 | | | here. | | | | |
| | X | IM | 15 | 15 | 18 | | | | | | | |
| | BA | R | 14 | | | 17 | | 1.771 | | | | |
| | BX | R | 14 | abon | | 17 | | | | | | |
| | | | | | | | | | | | | |

NOTE: Although both formats F2.1 and F3.2 exist for "LDB R,IM", the assembler always uses the short format, F3.2. In this case, the "src field" in the instruction format encoding contains the source operand.

| EXAMPLES: | LD | R0,R1 | !Register R0 is loaded with the contents of register R1! |
|-----------|-----|------------|---|
| | LD | R0,#20 | !Register R0 is loaded with the value 20! |
| | LD | R0,20 | !Register R0 is loaded with the contents of the word at location 20! |
| | LD | RO,@Rl | !If register Rl contains 20, then register R0 is loaded with the contents of the word at location 20! |
| | LDB | @Rl,RHO | !If register Rl contains 20, then the byte at location 20 is loaded with the contents of register RHO! |
| | LD | R4,8(R1) | !If register Rl contains 20, then register R4 is loaded with the contents of the word at location 28 (addressed by the sum of the address 8 and the contents of the index register R1)! |
| | LDL | RR4,R1(#8) | !If register Rl contains 20, then register RR4 is loaded with the con- tents of the long word at location 28 (addressed by the sum of the base address contained in register Rl and the displacement 8)! |
| | LD | Rl(R2),R4 | !If register Rl contains 20 and register R2 contains 8, then the word at location 28 (addressed by the sum of the base address contained in Rl and the contents of the index register R2) is loaded with the contents of register R4! |
| | | | |

LDA Load Address

LDA dst,src dst add dst: R src: DA, X, BA, BX INSTRUCTION FORMAT: (F2.1) mode dst src DA (src field=0) 01 R mode dst 0 1 1 STC 01 R X (src field<>0) (F4.4)dst src src det BA (src field<>0) 1 1 0 0 0 R displacement (F4.3)dst src det BX (src field<>0) 1 1 R 0 0 0 0 0 0 OPERATION: dst <- ADDRESS src The address of the source operand is computed and loaded into the destination. The contents of the source are not affected. The address computation follows the rules for addressing mode arithmetic. The destination is a word register in nonsegmented mode, and a register pair in segmented mode (the segmented address loaded into the destination has an undefined value in all "reserved" bits). FLAGS: No flags affected CYCLES: Address src NS SS SL 13 15 00 89 191 DA 12 13 13 16 X BA 15 BX 15 !In nonsegmented mode, register R4 is EXAMPLES: LDA R4, STRUCT loaded with the nonsegmented address of the location named STRUCT! LDA RR2,8(R4) !In segmented mode, if index register R4 contains %0020, then register RR2 is loaded with the segmented address %00000028 (segment 0, offset %0028)! !In segmented mode, if base register LDA RR2, RR4(#8) RR4 contains %01000020, then register RR2 is loaded with the segmented address %01000028 (segment 1,

offset %0028)!

LDAR Load Address Relative

LDAR dst,src

dst: R

INSTRUCTION FORMAT: (F4.4)

0 0 1 1 0 1 0 0 0 0 0 0 dst

displacement

 $\frac{\text{dst}}{R}$ $\frac{\text{src}}{RA}$

OPERATION:

dst <- ADDRESS src

The address of the source operand is computed and loaded into the destination. The contents of the source are not affected. The destination is a word register in nonsegmented mode, and a register pair in segmented mode (the segmented address loaded into the destination has all "reserved" bits cleared to zero).

FLAGS: No flags affected

CYCLES: Address

15

NOTE: The relative addressing mode is calculated by adding the

displacement in the instruction to the updated value of the program counter (PC) to derive the address. The updated PC value is taken to be the address of the instruction byte following the LDAR instruction, while the displacement is a 16-bit signed value in the range -32768 to +32767. The assembler automatically calculates the displacement by subtracting the PC value of the following

instruction from the address given by the programmer.

EXAMPLE: LDAR R2,DATA !Register R2 is loaded with the

nonsegmented address of the

location named DATA!

LDCTL **Load Control Register**

dst,src dst: R, CTLR LDCTL src: R, CTLR LDCTLB INSTRUCTION FORMAT: (F8.1)dst R CTLR B: 1 0 0 0 1 1 0 0 (F8.2)dst src R CTLR

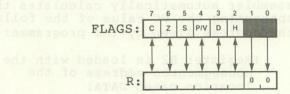
OPERATION: LDCTLB FLAGS, R FLAGS(2:7) <- src(2:7)

> The contents of the source (a byte register) loaded into the FLAGS register. The lower two bits of the FLAGS register and the entire source register are unaffected.

LDCTLB R, FLAGS dst(2:7) <- FLAGS(2:7) dst(0:1) <- 0

The contents of the upper six bits of the FLAGS register are loaded into the destination (a byte register). The lower two bits of the destination register are cleared to zero. The FLAGS register is unaffected.

Shaded areas are reserved



FLAGS:

LDCTLB FLAGS, R

C: Loaded from source Z: Loaded from source

S: Loaded from source P/V: Loaded from source

D: Loaded from source

H: Loaded from source

LDCTLB R, FLAGS No flags affected CYCLES: 7

NOTE: This is the only Load Control Register

instruction which is not privileged, thus it can be

used in Normal or System mode.

EXAMPLE: If the C, S and H flags are set (=1),

all other flags are clear (=0), the

statement

LDCTLB RHO, FLAGS

will leave the value %A4 (10100100) in RHO.

LDCTI.

Load Control Register

OPERATION:

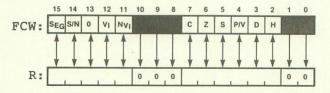
LDCTL FCW,R FCW(2:7) <- src(2:7) FCW(11:15) <- src(11:15)

The contents of the source (a word register) are loaded into the Flags and Control Word (FCW) register. Bits 0, 1, 8, 9 and 10 of the FCW register and the entire source register are unaffected.

LDCTL R,FCW dst(2:7) <- FCW(2:7) dst(11:15) <- FCW(11:15) dst(0:1) <- 0 dst(8:10) <- 0

The contents of the FCW register are loaded into the destination (a word register). Bits 0, 1, 8, 9 and 10 of the destination register are cleared to zero. The FCW register is unaffected.

Shaded areas are reserved



FLAGS:

LDCTL FCW, R

C: Loaded from source

Z: Loaded from source

S: Loaded from source P/V: Loaded from source

D: Loaded from source

H: Loaded from source

LDCTL R,FCW No flags affected

LDCTL Load Control Register

CYCLES:

7

NOTE:

This is a privileged instruction.

EXAMPLE:

If register R7 contains %D8FC (11011000111111100), the statement

LDCTL FCW.R7

will leave the value %D8FC in the FCW register (Segmentation and System modes are set, interrupts are enabled, and all the flags are set).

LDCTL

Load Control Register

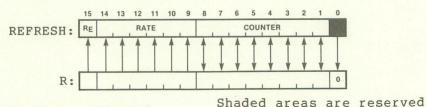
INSTRUCTION FORMAT: (F8.1) dst src R CTLR W: 0 1 1 1 1 1 0 1 src 0 1 1 (F8.2) dst src CTLR R W: 0 1 1 1 1 1 0 1 dst REFRESH, R OPERATION: LDCTL

REFRESH(1:15) <- src(1:15)

The contents of the source (a word register) are loaded into the REFRESH register. Bit 0 of the REFRESH register and the entire source register are unaffected.

LDCTL R,REFRESH
dst(1:8) <- REFRESH(1:8)
dst(0) <- 0
dst(9:15) <- undefined</pre>

The contents of the counter field (bits 1 to 8) of the REFRESH register are loaded into the destination (a word register). Bit 0 of the destination register is cleared to zero, and the upper seven bits are undefined. The REFRESH register is unaffected.



FLAGS:

No flags affected

CYCLES:

7

NOTE:

This is a privileged instruction.

EXAMPLE:

If register RO contains the value %8600 (10000110000000000), the statement

LDCTL REFRESH, RO

will leave the value %8600 in the REFRESH register (refresh is enabled, the rate is 3, and the counter starts at zero).

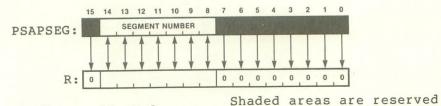
OPERATION:

LDCTL PSAPSEG,R PSAPSEG(8:14) <- src(8:14)

The contents of the source (a word register) are loaded into the Program Status Area Pointer Segment (PSAPSEG) register. Bits 0, 1, 2,...,7, and 15 of the PSAPSEG register and the entire source register are unaffected.

LDCTL R,PSAPSEG dst(8:14) <- PSAPSEG(8:14) dst(0:7) <- 0 dst(15) <- 0

The contents of the PSAPSEG register are loaded into the destination (a word register). Bits 0, 1, 2,...,7, and 15 of the destination register are cleared to zero. The PSAPSEG register is unaffected.



FLAGS: No flags affected

CYCLES: 7

NOTES: This is a privileged instruction.

The PSAPSEG register may not be used in the nonsegmented Z8002 version. In the segmented Z8001 version, care must be exercised when changing the two PSAP register values so that an interrupt occurring between the changing of PSAPSEG and PSAPOFF is handled correctly.

LDCTI.

Load Control Register

EXAMPLE: If register R12 contains %0200, the statement

LDCTL PSAPSEG, R12

will leave the value %0200 (segment 2) in the PSAPSEG register.

(8:14)

the destination register are The PSAPSEG register is

D. C. S. C. A. C. B. C. ST. III

p d d d d d d d d

r may not be us

t values so the changing

INSTRUCTION FORMAT: (F8.1)

W: 0 1 1 1 1 1 0 1 src 1 1 0 1 CTLR R

(F8.2)

W: 0 1 1 1 1 1 0 1 dst 0 1 0 1 R CTLR

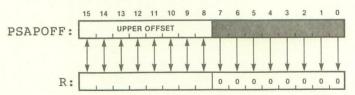
OPERATION: LDCTL PSAPOFF, R

PSAPOFF(8:15) <- src(8:15)

The contents of the source (a word register) are loaded into the Program Status Area Pointer Offset (PSAPOFF) register. Bits 0, 1, 2,...,7 of the PSAPOFF register and the entire source register are unaffected.

LDCTL R,PSAPOFF
dst(8:15) <- PSAPOFF(8:15)
dst(0:7) <- 0</pre>

The contents of the PSAPOFF register are loaded into the destination (a word register). Bits 0, 1, 2,...,7 of the destination register are cleared to zero. The PSAPOFF register is unaffected.



Shaded areas are reserved

FLAGS:

No flags affected

CYCLES:

7

NOTES:

This is a privileged instruction.

In the nonsegmented Z8002 version, the mnemonic "PSAP" should be used in the assembly language statement, and indicates the same control register as the mnemonic "PSAPOFF".

In the segmented Z800l version, care must be exercised when changing the two PSAP register values so that an interrupt occurring between the changing of PSAPSEG and PSAPOFF is handled correctly.

LDCTI.

Load Control Register

EXAMPLE:

If the PSAPOFF register contains %0100, non wormpusses

the statement

LDCTL R13, PSAPOFF

will leave the value %0100 in register R13.

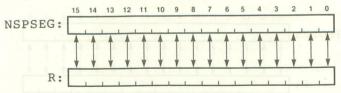
158

OPERATION: LDCTL NSPSEG,R

The contents of the source (a word register) are loaded into the Normal Stack Pointer Segment (NSPSEG) register. In segmented mode, the NSPSEG register is R14 in Normal mode, and contains the segment number of the normal processor stack pointer. (In nonsegmented mode, R14 is not used as part of the normal processor stack pointer.)

LDCTL R,NSPSEG dst <- NSPSEG

The contents of the NSPSEG register are loaded into the destination (a word register). The NSPSEG register is unaffected.



FLAGS: No flags affected

CYCLES: 7

NOTES: This is a privileged instruction.

The NSPSEG register may not be used in the nonsegmented Z8002 version.

EXAMPLE: If register Rl2 contains %7F00, the statement

LDCTL NSPSEG, R12

will leave the value %7F00 (segment %7F) in the NSPSEG register.

LDCTL Load Control Register

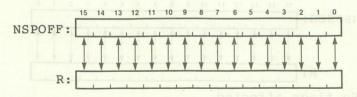
INSTRUCTION FORMAT: (F8.1) dst src CTLR R W: 0 1 1 1 1 1 0 src (F8.2)dst src CTLR W: 0 1 1 1 R 1 1 0 OPERATION: LDCTL NSPOFF, R

PERATION: LDCTL NSPOFF,R NSPOFF <- src

The contents of the source (a word register) are loaded into the Normal Stack Pointer Offset (NSPOFF) register. In segmented mode, the NSPOFF register is R15 in Normal mode, and contains the offset part of the normal processor stack pointer. In nonsegmented mode, R15 is the entire normal processor stack pointer. The source register is unaffected.

LDCTL R,NSPOFF

The contents of the NSPOFF register are loaded into the destination (a word register). The NSPOFF register is unaffected.



FLAGS:

No flags affected

CYCLES:

7

NOTES:

This is a privileged instruction.

In the nonsegmented Z8002 version, the mnemonic "NSP" should be used in the assembly language statement, and indicates the same control register

as the mnemonic "NSPOFF".

EXAMPLE:

If the NSPOFF register contains %F002, the statement

LDCTL R13,NSPOFF

will leave the value %F002 in register R13.

LDD dst.src.r LDDB

dst: IR sadd

INSTRUCTION FORMAT: (F6.4)

| W/B: | 1 | 0 | 1 | 1 | 1 | 0 | 1 | W/B | src | . 1 | 0 | 0 | 1 |
|------|---|---|---|---|---|---|---|-----|-----|-----|---|---|---|
| | 0 | 0 | 0 | 0 | | | r | | dst | 1 | 0 | 0 | 0 |

dst src r IR IR R

OPERATION: dst <- src

AUTODECREMENT dst and src {-1 if byte, -2 if word} r < -r - 1

This instruction is used for block transfers of strings of data. The contents of the location addressed by the source register are loaded into the location addressed by the destination register. The source and destination registers are then decremented by one if LDDB, or by two if LDD, thus moving the pointers to the previous elements in the strings. The word register specified by "r" (used as a counter) is then decremented by one.

- FLAGS: C: Unaffected
 - Z: Undefined S: Unaffected

 - V: Set if the result of decrementing r is zero; cleared
 - otherwise
 - D: Unaffected on and postmomeroes to doolte on
 - H: Unaffected work and the the transport at release

Word/Byte and phinals wasaabba vaomam javol ame 20 of bors appoints out to ago this deadold

EXAMPLE:

If register Rl contains %202A, register R2 contains %404A, the word at location %404A contains %FFFF, and register R3 contains 5, the statement

LDD @R1, @R2, R3

will leave the value %FFFF in location %202A, the value %2028 in Rl, the value %4048 in R2, and the value 4 in R3.

Load, Decrement and Repeat

LDDR dst,src,r

dst: IR src: IR

INSTRUCTION FORMAT: (F6.4)

W/B: 1 0 1 1 1 0 1 W/B src 1 0 0 1 1 0 0 0 1 dst 0 0 0 0 0

dst src r IR IR R

OPERATION:

dst <- src AUTODECREMENT dst and src $\{-1 \text{ if byte, } -2 \text{ if word}\}$ r <- r - 1

repeat until r = 0

This instruction is used for block transfers of strings of data. The contents of the location addressed by the source register are loaded into the location addressed by the destination register. The source and destination registers are then decremented by one if LDDRB, or by two if LDDR, thus moving the pointers to the previous elements in the strings. The word register specified by "r" (used as a counter) is then decremented by one. The entire operation is repeated until the result of decrementing r is zero. This instruction can transfer from 1 to 65536 bytes or 32768 words (the value for r must not be greater than 32768 for LDDR).

The effect of decrementing the pointers during the transfer is important if the source and destination strings overlap with the source string starting at a lower memory address. Placing the pointers at the highest address of the strings, and decrementing the pointers, ensures that the source string will be copied without destroying the overlapping area.

FLAGS:

C: Unaffected dada and ad anta

Z: Undefined
S: Unaffected

V: Set

D: Unaffected of all Mada sulay and avail live

H: Unaffected at 840M and av add al8 at 8500

CYCLES:

Word/Byte

11+9*n (n=number of data elements transferred)

Load, Decrement and Repeat

NOTE:

This instruction can be interrupted after each execution of the basic operation. The program counter value of the start of this instruction is saved before the interrupt request is accepted, so that the instruction can be properly resumed. Seven more cycles should be added to this instruction's execution time for each interrupt request that is accepted.

EXAMPLE:

If register R1 contains %202A, register R2 contains %404A, the words at locations %4040 through %404A all contain %FFFF, and register R3 contains 5, the statement

LDDR @R1, @R2, R3

will leave the value %FFFF in the words at locations %2020 through %202A, the value %201E in R1, the value %403E in R2, and 0 in R3.

Load and Increment

LDI dst,src,r

dst: IR src: IR

INSTRUCTION FORMAT: (F6.4)

7/B: 1 0 1 1 1 0 1 W/B SrC 0 0 0 1 0 1 0 0 0 0 1

 $\frac{\text{dst}}{\text{IR}} \quad \frac{\text{src}}{\text{IR}} \quad \frac{\text{r}}{\text{R}}$

OPERATION:

dst <- src AUTOINCREMENT dst and src $\{+1 \text{ if byte, } +2 \text{ if word}\}$ r <- r - 1

This instruction is used for block transfers of strings of data. The contents of the location addressed by the source register are loaded into the location addressed by the destination register. The source and destination registers are then incremented by one if LDIB, or by two if LDI, thus moving the pointers to the next elements in the strings. The word register specified by "r" (used as a counter) is then decremented by one.

FLAGS: C: Unaffected

Z: Undefined

S: Unaffected

V: Set if the result of decrementing r is zero; cleared

otherwise D: Unaffected H: Unaffected

CYCLES:

Word/Byte 20

EXAMPLE:

This instruction is used in a "loop" of instructions which transfers a string of data from one location to another, but an intermediate operation on each data element is required. The following sequence transfers a string of 80 bytes, but tests for a special value (%0D, an ASCII return character) which terminates the loop if found:

LD R3,#80 !Initialize counter!
LD R1,#DSTBUF !Load start addresses!
LD R2,#SRCBUF

LOOP:

CPB @R2,#%0D !Check for return character!

JR EQ,DONE !Exit loop if found!

LDIB @R1,@R2,R3 !Transfer next byte!

JR NOV,LOOP !Repeat until counter=0!

DONE:

Load, Increment and Repeat

LDIR dst, src, r LDIRB

dst: IR src: IR

(F6.4)INSTRUCTION FORMAT:

0 0 dst 0 0 0 0

dst src IR IR

OPERATION:

dst <- src AUTOINCREMENT dst and src {+l if byte, +2 if word} r <- r - 1 repeat until r = 0

This instruction is used for block transfers of strings of data. The contents of the location addressed by the source register are loaded into the location addressed by the destination register. The source and destination registers are then incremented by one if LDIRB, or by two if LDIR, thus moving the pointers to the next elements in the strings. The word register specified by "r" (used as a counter) is then decremented by one. The entire operation is repeated until the result of decrementing r is zero. This instruction can transfer from 1 to 65536 bytes or 32768 words (the value for r must not be greater than 32768 for LDIR).

The effect of incrementing the pointers during the transfer is important if the source and destination strings overlap with the source string starting at a higher memory address. Placing the pointers at the lowest address of the strings, and incrementing the pointers, ensures that the source string will be copied without destroying the overlapping area.

FLAGS:

C: Unaffected

Z: Undefined S: Unaffected

V: Set

D: Unaffected H: Unaffected

CYCLES:

Word/Byte 11+9*n (n=number of data elements transferred)

LDIR

Load, Increment and Repeat

NOTE:

This instruction can be interrupted after each execution of the basic operation. The program counter value of the start of this instruction is saved before the interrupt request is accepted, so that the instruction can be properly resumed. Seven more cycles should be added to this instruction's execution time for each interrupt request that is accepted.

EXAMPLE:

The following sequence of instructions can be used to copy a buffer of 512 words (1024 bytes) from one area to another. The pointers to the start of the source and destination buffers are set, the number of words to transfer is set, and then the transfer is accomplished.

LD R1, #DSTBUF LD R2, #SRCBUF LD R3, #512 LDIR @R1, @R2, R3 LDK dst,src

dst: R src: IM

INSTRUCTION FORMAT: (F1.2)

W: mode src 1 1 1 1 0 1

dst_msrc MoIn MATERIA mode 10 R

OPERATION:

dst <- src {src is 0 to 15}

The source operand (a constant value) is loaded into the destination. The source is a value from 0 to 15 which is loaded into the low-order four bits of the destination with the high-order twelve bits cleared to zeros.

FLAGS:

No flags affected wromen most aredstpen

CYCLES:

Word 5

NOTE:

The "src field" in the instruction format encoding contains the source operand. The "src field" values range from 0 to 15 corresponding to the source values

0 to 15.

EXAMPLE:

LDK R3, #9

!Register R3 is loaded with the

value 9!

Load Multiple

LDM dst, src, num

dst: R, IR, DA, X src: R, IR, DA, X

num: IM

INSTRUCTION FORMAT: (F6.1)

| m | od | е | 0 | 1 | 1 | 1 | 0 | 0 | | S | rc | | 0 | 0 | 0 | , |
|---|-----|---|---|---|---|---|----|---|---|---|----|---|---|----|---|---|
| 0 | - 3 | 0 | 0 | 0 | | d | st | | 0 | 0 | 0 | 0 | | nu | m | |

| mode | dst | sro | | |
|------|-----|-----|------|----------|
| 00 | R | IR | | |
| 01 | R | DA | (src | field=0) |
| 01 | R | X | (src | field<>0 |

| 0 | mode | 0 | 1 | 1 | 1 1 | 0 | 0 | h | d | st | | 1 | 0 | 0 | 1 |
|---|------|---|---|-----|-----|---|---|---|---|----|---|---|----|----|---|
| | 0 0 | 0 | 0 | 132 | sr | С | | 0 | 0 | 0 | 0 | | nı | ım | |

| mode | dst | src | |
|------|-----|--------|-----------|
| 00 | IR | R | |
| 01 | DA | R (dst | field=0) |
| 01 | X | R (dst | field(>0) |

Registers from Memory Rdst, Rdst+num-l <- src

The contents of num source words are loaded into the destination register and the following (num-1) registers. The contents of the source(s) are not affected. The value of num can vary from 1 to 16, and the registers are loaded in increasing order, with RO following R15.

Memory from Registers dst <- Rsrc,Rsrc+num-1</pre>

The contents of num word registers including the source register and the following (num-1) registers are loaded into the destination words. The contents of the source(s) are not affected. The value of num can vary from 1 to 16, and the registers are loaded in increasing order, with RO following R15.

FLAGS:

No flags affected

CYCLES:

| dst | src | NS | Word | SL | | | |
|-----|-----|--------|--------|--------|-----------|----|------------|
| R | IR | 11+3*n | | | (n=number | of | registers) |
| R | DA | 14+3*n | 15+3*n | 17+3*n | | | |
| R | X | 15+3*n | 15+3*n | 18+3*n | | | |
| IR | R | 11+3*n | | | | | |
| DA | R | 14+3*n | 15+3*n | 17+3*n | | | |
| X | R | 15+3*n | 15+3*n | 18+3*n | | | |

NOTES:

The "num field" in the instruction format encoding contains the num operand. The "num field" values range from 0 to 15 corresponding to the num values 1 to 16.

The starting address of the block of memory where the registers are loaded from or to is computed once at the start of execution, and incremented by two for each register loaded. If the original address computation involved a register, the register's value will not be affected by the address incrementation during execution.

EXAMPLE: If register R5 contains 5, R6 contains %0100, and R7 contains 7, the statement

LDM @R6,R5,#3

will leave the values 5, %0100, and 7 at word locations %0100, %0102, and %0104, respectively, and none of the registers will be affected.

Load Program Status

LDPS src src: IR, DA, X

INSTRUCTION FORMAT:

(F2.3)

1 1 1 0 0 1 0 n mode IR 00 01 DA (src field=0) 01 X (src field<>0)

OPERATION:

PS <- src

The contents of the source operand are loaded into the Program Status (PS), loading the Flags and Control Word (FCW) and the program counter (PC). The new value of the FCW does not become effective until the next instruction, so that the status pins will not be affected by the new control bits until after the LDPS instruction execution is completed. The next instruction executed is that addressed by the new contents of the PC. The contents of the source are not affected.

This instruction is used to set the Program Status of a program, and is particularly useful for setting the System/Normal mode of a program to Normal mode, or for running a nonsegmented program in the segmented Z8001 version. The PC segment number is not affected by the LDPS instruction in nonsegmented mode.

The format of the source operand (Program Status block) depends on the current Segmentation mode (not on the version of the Z8000), and is illustrated in the following figure:

NONSEGMENTED FCW PC

LOW ADDRESS

SEGMENTED FCW PC SEG NO PC OFFSET

HIGH ADDRESS

(shaded area is reserved -- must be zero)

Load Program Status

FLAGS: C: Loaded from source
Z: Loaded from source
S: Loaded from source
P/V: Loaded from source
D: Loaded from source
H: Loaded from source

NOTE: This is a privileged instruction.

EXAMPLE: In the nonsegmented Z8002 version, if the program counter contains %2550, register R3 contains %5000, location %5000 contains %1800, and location %5002 contains %A000, the statement

LDPS @R3

value in the location named DATA1

will leave the value %A000 in the program counter, and the FCW value will be %1800 (indicating Normal mode, interrupts enabled, and all flags cleared).

Load Relative

LDR dst, src dst: R, RA LDRB src: R, RA LDRI. INSTRUCTION FORMAT: (F4.4)dst src 0 0 W/B 0 dst W/B: 0 n RA displacement L: 1 0 0 det displacement (F4.2)dst src 1 W/B 0 0 src W/B: RA R displacement 1 1 0 0 0 L: displacement OPERATION: dst <- src The contents of the source operand are loaded into the destination. The contents of the source are not affected. FLAGS: No flags affected Word/Byte CYCLES: Long 17 NOTES: The relative addressing mode is calculated by adding the displacement in the instruction to the updated value of the program counter (PC) to derive the operand's address. The updated PC value is taken to be the address of the instruction byte following the LDR, LDRB, or LDRL instruction, while the displacement is a 16-bit signed value in the range -32768 to +32767. The assembler auto-

Status pin information during the access to memory for the data operand will be IFn instead of $\overline{\text{MREQ}}$ because the address is relative to PC.

matically calculates the displacement by subtracting the PC value of the following instruction from the address

EXAMPLE: LDR R2,DATA

given by the programmer.

!Register R2 is loaded with the value in the location named DATA!

MBIT

INSTRUCTION FORMAT: (F9.3)

0 1 1 1 1 0 1 1 0 0 0 0 1 0 1 0

OPERATION: S <- NOT UI

This instruction is used to synchronize multiple processors' exclusive access to shared hardware resources. The multi-micro input pin $(\overline{\mu I})$ is tested, and the S flag is cleared if the pin is one (active state = low voltage); otherwise, the S flag is set, indicating that the pin is zero (inactive state = high voltage).

After the MBIT instruction is executed, the S flag can be used to determine whether a requested resource is available or not. If the S flag is clear, then the resource is not available; if the S flag is set, then the resource is available for use by this CPU.

FLAGS: C: Unaffected

Z: Undefined

S: Set if uI is zero; cleared otherwise

V: Unaffected
D: Unaffected
H: Unaffected

CYCLES: 7

NOTE: This is a privileged instruction.

EXAMPLE: The following sequence of instructions can be used to wait for the availability of a

resource.

LOOP:

MBIT | !Test multi-micro input!

JR PL,LOOP !Repeat until resource is available!

AVAILABLE:

MREQ Multi-Micro Request

MREO dst dst: R INSTRUCTION FORMAT: (F8.2) dst 1 1 0 1 0 1 1 1 1 0 1 1 R OPERATION: z < -0if $\overline{uI} = 1$ then S < - 0This instruction is 0 < -0 at nothern self. else $\overline{\mu 0}$ <- 1 repeat dst <- dst - 1 pall 2 add salwyaddo (until dst = 0 adada ayddaa symbol ones at min if $\overline{\mu I}$ = 1 then S <- 1 else S <- 0 <u>uO</u> <- 0 sharruft->nZis executed, the S flag

This instruction is used to synchronize multiple processors' exclusive access to shared hardware resources. A request for a resource is signalled through the multi-micro input and output pins (μI and μO), with the S and Z flags indicating the availability of the resource after the MREQ instruction has been executed.

First, the Z flag is cleared. Then the $\overline{\mu I}$ pin is tested. If the $\overline{\mu I}$ pin is one (active state = low voltage), the S flag is cleared and the $\overline{\mu O}$ pin is cleared to zero (inactive state = high voltage), thus indicating that the resource is not available. Instruction execution is finished in this case.

If the $\overline{\mu I}$ pin is zero, indicating that the resource may be available, a sequence of machine operations occurs. First, the $\overline{\mu O}$ pin is set to one (active state = low voltage), signalling a request by the CPU for the resource. Next, a finite delay to allow for propagation of the signal to other processors is accomplished by repeatedly decrementing the contents of the destination (a word register) until its value is zero. Then the $\overline{\mu I}$ pin is tested to determine whether the request for the resource was acknowledged. If the $\overline{\mu I}$ pin is one (active state = low voltage), the S flag is set to one indicating that the resource is available and access is granted. If the $\overline{\mu I}$ pin is zero (inactive state = high voltage), the S flag is

cleared to zero, and the $\overline{\mu O}$ pin is cleared to zero (inactive state = high voltage), indicating that the request was not granted. Finally, in either case, the Z flag is set to one, indicating that the original test of the $\overline{\mu I}$ pin caused a request to be made.

| - | flag | - | lag | - | Indicates |
|----------|--------|---------|---------|---------|--|
| 6d 4) | shareO | 0 0 | access(| 0 1 6 1 | Request not signalled (resource not available) |
| | O | 1 to | beau a | 0 | Request not granted (resource not available) |
| | 1 | 1 | | | Request granted (resource available) |

FLAGS: C: Unaffected

Z: Set if request was signalled; cleared otherwise

S: Set if request was signalled and granted;

cleared otherwise dank bepolity

TRY:

CYCLES:

12 (request not signalled)

12+7*n (request signalled, n=number

of times destination is decremented)

NOTE:

This is a privileged instruction.

EXAMPLE:

LD R0,#50 MREO R0 !Allow for propagation delay! !Multi-micro request with

delay in register RO!

JR MI,AVAILABLE
JR Z,NOT GRANTED

NOT AVAILABLE: . !Resource not available!

۰

NOT GRANTED: . !Request not granted!

JR TRY !Try again after awhile!

AVAILABLE: . !Use resource!

•

MRES !Release resource!

MRES Multi-Micro Reset

MRES - 10 at aid Of add bas cores of betseld

INSTRUCTION FORMAT: (F9.3)

OPERATION: $\overline{\mu O}$ <- 0

This instruction is used to synchronize multiple processors' exclusive access to shared hardware resources. The multi-micro output pin $(\mu \overline{0})$ is cleared to zero (inactive state = high voltage).

Resetting $\overline{\mu O}$ to zero is used to indicate that a resource controlled by the CPU is available for use

by other processors.

FLAGS: No flags affected

CYCLES: 5 Man have for each family as well as many as the second of the

NOTE: This is a privileged instruction.

EXAMPLE: MRES !Signal that resource controlled

by this CPU is available to

other processors!

MSET

INSTRUCTION FORMAT: (F9.3)

0 1 1 1 1 0 1 1 0 0 0 0 1 0 0 0

OPERATION: $\overline{\mu O}$ <- 1

This instruction is used to synchronize multiple processors' exclusive access to shared hardware resources. The multi-micro output pin $(\overline{\mu 0})$ is set to one (active state = low voltage).

Setting $\overline{u0}$ to one is used either to indicate that a resource controlled by the CPU is not available to other processors, or to signal a request for a resource controlled by some other processor.

FLAGS: No flags affected

CYCLES: 5

NOTE: This is a privileged instruction.

EXAMPLE: MSET !Signal that controlled resource is not available to other processors!

MIII.T Multiply

MULT dst, src MULTL

dst: R

src: R, IM, IR, DA, X

INSTRUCTION FORMAT: (F2.1)

| | | | | | | | | | | | | La | mode | dst | src | | |
|----|------|---|----------|---|---|---|---|---|-----|---|---|-----|--------|-----|-----|------|-----------|
| W: | mode | 0 | 1 | 1 | 0 | 0 | 1 | | sre | C | | dst | 10 | R | R | | |
| | | | <u> </u> | _ | _ | | _ | _ | | | | 1 1 | 00 | R | IM | (src | field=0) |
| L: | mode | 0 | 1 | 1 | 0 | 0 | 0 | Т | sre | С | T | dst | 00 | R | | | field<>0) |
| | | | <u> </u> | _ | | _ | _ | | | | | 1 | 01 | R | | | field=0) |
| | | | | | | | | | | | | | 01 | R | X | (src | field<>0) |

OPERATION:

Word

dst,dstyl <- dstyl * src

Long

dst,dstyl,dsty2,dsty3 <- dsty2,dsty3 * src,srcvl

The low-order half of the destination operand (multiplicand) is multiplied by the source operand (multiplier) and the product is stored in the destination. The contents of the source are not affected. Both operands are treated as signed, two's complement integers. For MULT, the destination is a register pair and the source is a word value; for MULTL, the destination is a register quadruple and the source is a long word value.

FLAGS:

- C: MULT set if product is less than -215 or greater than or equal to 215; cleared otherwise; MULTL - set if product is less than 231 or greater than or equal to 231; cleared otherwise
- Z: Set if the result is zero; cleared otherwise
- S: Set if the result is negative; cleared otherwise
- V: Cleared D: Unaffected
- H: Unaffected

| CYCLES: | src | Word | | Long | |
|---------|-------------|----------|---------|---------|-----------|
| | daysayadhaa | NS SS SL | NS | SS | SL |
| | R | 70 | 282+7*n | | |
| | IM | 70 | 282+7*n | (1, ±=) | ON FORMAT |
| | IR | 70 | 282+7*n | | |
| | DA | 71 72 74 | 283+7*n | 284+7*n | 286+7*n |
| | X | 72 72 75 | 284+7*n | 284+7*n | 287+7*n |

(n=number of bits equal to one in the absolute value of the low-order 16 bits of the destination operand)

(Multiplier is zero) MEG and ABO for MEGE are replaced by them

| R | 18 | | na fo | 30 | | |
|----|----|----|--------|------------|----|----|
| IM | 18 | | eeg on | 30 | | |
| IR | 18 | | | 30 | | |
| DA | 19 | 20 | 22 | 31 0 2 9 3 | 32 | 34 |
| X | 20 | 20 | 22 | 32 | 32 | 35 |
| | | | | | | |

NOTE: For proper instruction execution, the "dst field" in the instruction format encoding must be even for MULT, and must be a multiple of 4 (0, 4, 8, 12) for MULTL. If the source operand in MULTL is a register, the "src field" must be even.

EXAMPLE:

If register RQO (composed of register pairs RRO and RR2) contains %2222222200000031 (RR2 contains decimal 49), the statement

MULTL ROO, #10

will leave the value %0000000000001EA (decimal 490) in RQO.

Negate

NEG dst NEGB

dst: R, IR, DA, X

mode dst

INSTRUCTION FORMAT: (F1.1)

W/B: mode 0 0 1 1 0 W/B

10 R 00 IR R 01 DA (dst field=0) 01 X (dst field<>0)

OPERATION: dst <- - dst

The contents of the destination is negated, that is, replaced by its two's complement value. Note that \$8000 for NEG and \$80 for NEGB are replaced by themselves since in two's complement representation the most negative number has no positive counterpart.

FLAGS: C: Cleared if the result is zero; set otherwise, which indicates a "borrow"

Z: Set if the result is zero; cleared otherwise

S: Set if the result is negative; cleared otherwise

V: Set if the result is %8000 for NEG, or %80 for NEGB; cleared otherwise

H: Unaffected

CYCLES: Word/Byte dst NS SS SL RR2) contains 822222222000-03 -- R 7 contains decima 12 ___ TR 15 16 18 DA X 16 16 19

EXAMPLE: If register R8 contains %051F, the statement

NEG R8

will leave the value %FAEl in R8.

NOP

INSTRUCTION FORMAT: (F9.3)

1 0 0 0 1 1 0 1 0 0 0 0 0 1 1 1

OPERATION:

No operation is performed. This instruction may be

used for timing delays.

FLAGS: No flags affected

CYCLES: 7

OR dst.src ORB

dst: R

src: R, IM, IR, DA, X

INSTRUCTION FORMAT: (F2.1)

| 111011 | | 10. | | | | | • | (1201) | | mode | dst | src | | |
|--------|------|-----|---|---|---|---|-----|--------|-----|------|-----|-----|------|-----------|
| W/B: | mode | 0 | 0 | 0 | 1 | 0 | W/B | src | dst | 10 | R | R | | |
| ., - | | _ | _ | _ | _ | _ | | | | 00 | R | IM | (src | field=0) |
| | | | | | | | | | | 00 | R | IR | (src | field<>0) |
| | | | | | | | | | | 01 | R | DA | (src | field=0) |
| | | | | | | | | | | 01 | R | X | (src | field<>0) |

OPERATION:

dst <- dst OR src

The source operand is logically ORed with the destination operand and the result is stored in the destination. The contents of the source are not affected. The OR operation results in a one bit being stored whenever either of the corresponding bits in the two operands is one; otherwise, a zero bit is stored.

FLAGS: C: Unaffected

Z: Set if the result is zero; cleared otherwise

S: Set if the most significant bit of the result is set;

cleared otherwise

P: OR - unaffected; ORB - set if parity of the result is even; cleared otherwise

D: Unaffected

H: Unaffected

CYCLES:

| src | Wo | rd/B | yte |
|-----|----|------|-----|
| | NS | SS | SL |
| R | 4 | | |
| IM | 7 | | |
| IR | 7 | | |
| DA | 9 | 10 | 12 |
| X | 10 | 10 | 13 |

EXAMPLE:

If register RL3 contains %C3 (11000011) and the source operand is the immediate value %7B (01111011), the statement

ORB RL3, #%7B

will leave the value %FB (11111011) in RL3.

OTDR dst,src,r

dst: IR src: IR

INSTRUCTION FORMAT: (F6.4)

 $\frac{\text{dst}}{\text{IR}} \quad \frac{\text{src}}{\text{IR}} \quad \frac{\text{r}}{\text{R}}$

OPERATION: dst <- src

AUTODECREMENT src {-1 if byte, -2 if word}
r <- r - 1
repeat until r = 0

This instruction is used for block output of strings of data. The contents of the memory location addressed by the source register are loaded into the I/O location addressed by the destination register. I/O addresses are always 16 bits. The source register is then decremented by one if OTDRB, or by two if OTDR, thus moving the pointer to the previous element in the string. The word register specified by "r" (used as a counter) is then decremented by one. The entire operation is repeated until the result of decrementing r is zero. This instruction can output from 1 to 65536 bytes or 32768 words (the value for r must not be greater than 32768 for OTDR).

FLAGS:

C: Unaffected

Z: Undefined

S: Unaffected

V: Set

D: Unaffected

H: Unaffected

CYCLES:

Word/Byte 11+10*n

(n=number of data elements transferred)

NOTES:

This is a privileged instruction.

This instruction can be interrupted after each execution of the basic operation. The program counter value of the start of this instruction is saved before the interrupt request is accepted, so that the instruction can be properly resumed. Seven more cycles should be added to this instruction's execution time for each interrupt request that is accepted.

OTDR

Output, Decrement and Repeat

EXAMPLE:

If register Rll contains %0FFF, register Rl2 contains %B006, and Rl3 contains 6, the statement

OTDR @R11,@R12,R13

will output the string of words from locations %AFFC to %B006 in descending order to "port" %0FFF. R12 will contain %AFFA, and R13 will contain 0. R11 will not be affected.

OTIR dst,src,r

dst: IR src: IR

INSTRUCTION FORMAT: (F6.4)

| W/B: | 0 | 0 | 1 | 1 | 1 | 0 | 1 | W/B | src | 0 | 0 | 1 | 0 |
|------|---|---|---|---|---|---|---|-----|-----|---|---|---|---|
| | 0 | 0 | 0 | 0 | | | r | | dst | 0 | 0 | 0 | 0 |

 $\frac{\text{dst}}{\text{IR}} \quad \frac{\text{src}}{\text{IR}} \quad \frac{\text{r}}{\text{R}}$

OPERATION: dst

dst <- src AUTOINCREMENT src $\{+1 \text{ if byte, } +2 \text{ if word}\}$ r <- r - 1

repeat until r = 0

This instruction is used for block output of strings of data. The contents of the memory location addressed by the source register are loaded into the I/O location addressed by the destination register. I/O addresses are always 16 bits. The source register is the incremented by one if OTIRB, or by two if OTIR, thus moving the pointer to the next element in the string. The word register specified by "r" (used as a counter) is then decremented by one. The entire operation is repeated until the result of decrementing r is zero. This instruction can output from 1 to 65536 bytes or 32768 words (the value for r must not be greater than 32768 for OTIR).

FLAGS: C: Unaffected

Z: Undefined

S: Unaffected

V: Set

D: Unaffected

H: Unaffected

CYCLES:

Word/Byte 11+10*n

(n=number of data elements transferred)

NOTES:

This is a privileged instruction.

This instruction can be interrupted after each execution of the basic operation. The program counter value of the start of this instruction is saved before the interrupt request is accepted, so that the instruction can be properly resumed. Seven more cycles should be added to this instruction's execution time for each interrupt request that is accepted.

EXAMPLE:

The following sequence of instructions can be used to output a string of bytes to the specified I/O "port". The pointers to the I/O port and the start of the source string are set, the number of bytes to output is set, and then the output is accomplished.

LD R1,#PORT
LD R2,#SRCBUF
LD R3,#LENGTH
OTIRB @R1,@R2,R3

dst: IR,DA dst.src OUT src: R OUTB . INSTRUCTION FORMAT: (F7.4)dst src STC W/B: 0 0 1 1 1 1 1 W/B det (F7.2)dst src DA R 0 1 1 0 W/B: 0 0 1 0 1 W/B src dst <- src OPERATION: The contents of the source operand are loaded into the destination. I/O addresses are always 16 bits. No flags affected FLAGS: CYCLES: dst Word/Byte 10 12 DA

If register R6 contains %5252, the statement

This is a privileged instruction.

OUT %1120,R6

NOTE:

EXAMPLE:

will output the value %5252 to the "port" %1120.

Output and Decrement

OUTD dst,src,r

dst: IR src: IR

INSTRUCTION FORMAT: (F6.4)

 $\frac{\text{dst}}{\text{IR}} \quad \frac{\text{src}}{\text{IR}} \quad \frac{\text{r}}{\text{R}}$

OPERATION:

dst <- src
AUTODECREMENT src {-l if byte, -2 if word}</pre>

This instruction is used for block output of strings of data. The contents of the memory location addressed by the source register are loaded into the I/O location addressed by the destination register. I/O addresses are always 16 bits. The source register is then decremented by one if OUTDB, or by two if OUTD, thus moving the pointer to the previous element in the string. The word register specified by "r" (used as a counter) is then decremented by one.

FLAGS:

C: Unaffected Z: Undefined

S: Unaffected

V: Set if the result of decrementing r is zero;

cleared otherwise

D: Unaffected
H: Unaffected

CYCLES:

Word/Byte

NOTE:

This is a privileged instruction.

EXAMPLE:

In segmented mode, if register R2 contains the I/O address \$0030, register RR6 contains \$12005552 (segment \$12, offset \$5552), the word at memory location \$12005552 contains \$1234, and register R8 contains \$1001, the statement

OUTD @R2,@RR6,R8

will output the value %1234 to "port" %0030, and leave the value %12005550 in RR6, and %1000 in R8. Register R2 will not be affected.

OUTI dst,src,r

1909 dst: IR

INSTRUCTION FORMAT: (F6.4)

 $\frac{\text{dst}}{\text{IR}}$ $\frac{\text{src}}{\text{IR}}$ $\frac{\text{r}}{\text{R}}$

OPERATION: dst <- src

AUTOINCREMENT src {+1 if byte, +2 if word}
r <- r - 1

This instruction is used for block output of strings of data. The contents of the memory location addressed by the source register are loaded into the I/O location addressed by the destination register. I/O addresses are always 16 bits. The source register is then incremented by one if OUTIB, or by two if OUTI, thus moving the pointer to the next element in the string. The word register specified by "r" (used as a counter) is then decremented by one.

FLAGS: C: Unaffected

Z: Undefined S: Unaffected

V: Set if the result of decrementing r is zero;

cleared otherwise

D: Unaffected H: Unaffected

CYCLES: Word/Byte

NOTE: This is a privileged instruction.

EXAMPLE: This instruction is used in a "loop"
of instructions which outputs a string of data, but
an intermediate operation on each element is
required. The following sequence outputs a string
of 80 ASCII characters (bytes) with the most
significant bit of each byte set or reset to
provide even parity for the entire byte.

LD Rl, #PORT !Load I/O address! !Load start of string! LD R2, #SRCSTART !Initialize counter! LD R3, #80 LOOP: TESTB !Test byte parity! @R2 JR PE, EVEN SETB @R2,#7 !Force even parity! EVEN: OUTIB @R1, @R2, R3 !Output next byte! !Repeat until counter=0! NOV, LOOP JR DONE:

POP dst,src

dst: R, IR, DA, X

mode dst src

INSTRUCTION FORMAT: (F2.1)

| mode | 0 | 1 | 0 | 1 | 1 | 1 | src | dst |
|------|---|---|---|---|---|---|-----|-----|
| | _ | | | _ | 1 | | | |

10 R IR
00 IR IR
01 DA IR (dst field=0)
01 X IR (dst field<>0)

OPERATION:

dst <- src

AUTOINCREMENT src {+2 if word, +4 if long}

The contents of the location addressed by the source register (a "stack pointer") are loaded into the destination. The source register is then incremented by a value which equals the size in bytes of the destination operand, thus removing the top element of the "stack" by changing the stack pointer. Any register except R0 (or RR0) can be used as a stack pointer.

FLAGS:

No flags affected

CYCLES:

| dst | | Word | | | Long | |
|-----|----|------|----|----|------|-------|
| 500 | NS | SS | SL | NS | SS | SL |
| R | 8 | | | 12 | - 22 | Ha-Oc |
| IR | 12 | | | 19 | | |
| DA | 16 | 16 | 19 | 23 | 23 | 26 |
| X | 16 | 16 | 19 | 23 | 23 | 26 |

NOTE:

For POPL, the same register must not be used in both the source and destination addressing mode designators.

EXAMPLE:

If register R12 (a "stack pointer") contains %1000, the word at location %1000 contains %0055, and register R3 contains %0022, the statement

POP R3,@R12

will leave the value %0055 in R3 and the value %1002 in R12.

Push

PUSH dst, src dst: IR PUSHI. src: R, IM, IR, DA, X INSTRUCTION FORMAT: (F2.2)mode dst src 10 TR R det 1 0 0 1 1 00 IR TR 01 DA (src field=0) dst TR T. e mode 1 0 0 0 1 src 07 IR X (src field<>0) (F5.1)dst mode src TM 0.0 TR W: mode dst 0 0 1 1 0 1

OPERATION:

AUTODECREMENT dst {-2 if word, -4 if long} dst <- src

The contents of the destination register (a "stack pointer") is decremented by a value which equals the size in bytes of the destination operand. Then the source operand is loaded into the location addressed by the updated destination register, thus adding a new element to the top of the "stack" by changing the stack pointer. Any register except RO (or RRO) can be used as a stack pointer.

FLAGS:

No flags affected

CYCLES:

| src | | Word | | | Long | |
|-----|----|------|---------|------|------|----|
| | NS | SS | SL | NS | SS | SL |
| R | 9 | | 112330 | 12 | | |
| IM | 12 | 6_00 | T 2 PUT | 9890 | PUE | |
| IR | 13 | | | 20 | | |
| DA | 14 | 14 | 17 | 21 | 21 | 24 |
| X | 14 | 14 | 17 | 21 | 21 | 24 |

NOTE:

For PUSHL, the same register must not be used in both the source and destination addressing mode designators.

EXAMPLE:

If register R12 (a "stack pointer") contains %1002, the word at location %1000 contains %0055, and register R3 contains %0022, the statement

PUSH @R12,R3

will leave the value %0022 in location %1000 and the value %1000 in R12.

RES dst,src RESB INSTRUCTION FORMAT: (F1.2) dst: R, IR, DA, X src: R, IM

| | | | | | | | | | | mode | dst | src | F 10 | |
|------|------|---|---|---|---|---|-----|-----------|-----------|------|-----|-----|------|-----------|
| W/B: | mode | 1 | 0 | 0 | 0 | 1 | W/B | dst | src | 10 | R | IM | | |
| , | | | | - | _ | | | - more of | MA AFRANC | 00 | IR | | | field<>0) |
| | | | | | | | | | | 01 | DA | IM | (dst | field=0) |
| | | | | | | | | | | 01 | X | IM | (dst | field<>0) |
| | | | | | | | | 100 21 | | | | | | |

0 0 1 W/B 0 0 0 0 0 0 0 0 0

dst src R

OPERATION: dst(src) < -0

0

0 0

dst

W/B: 0

Clears the specified bit within the destination operand without affecting any other bits in the destination. The source (the bit number) can be specified as either an immediate value, or as a word register which contains the value. In the second case, the destination operand must be a register, and the source operand must be RO through R7 for RESB, or R0 through R15 for RES. The bit number is a value from 0 to 7 for RESB, or 0 to 15 for RES, with 0 indicating the least significant bit.

FLAGS: No flags affected

Word/Byte CYCLES: src NS SS R IM 4 11 TR IM 13 14 16 DA TM 17 X IM 14 14 R R 10

NOTE: Only the lower four bits of the source operand are used to specify the bit number for RES, while only the lower three bits of the source operand are used with RESB. When the source operand is an immediate value, the "src field" in the instruction format encoding contains the bit number in the lowest four bits for RES, or the lowest three bits for RESB.

If register RL3 contains %B2 (10110010), the statement EXAMPLE: RESB RL3,#1

will leave the value %BO (10110000) in RL3.

RESFLG Reset Flag

RESFLG flag

flag: C, Z, S, P, V

INSTRUCTION FORMAT: (F9.1)

1 0 0 0 1 1 0 1 C z S P/V 0 0 1 1

OPERATION: FLAGS(4:7) <- FLAGS(4:7) AND NOT instruction(4:7)

Any combination of the C, Z, S, P or V flags are cleared to zero if the corresponding bit in the instruction is one. If the corresponding bit in the instruction is zero, the flag will not be affected. All other bits in the FLAGS register are unaffected. Note that the P and V flags are represented by the same bit. There may be one, two, three or four operands in the assembly language statement, in any order.

FLAGS: C: Cleared if specified; unaffected otherwise

Z: Cleared if specified; unaffected otherwise
S: Cleared if specified; unaffected otherwise
P/V: Cleared if specified; unaffected otherwise

D: Unaffected H: Unaffected

CYCLES: 7

EXAMPLE: If the C, S, and V flags are all set (=1),

and the Z flag is clear (=0), the statement

RESFLG C. V

will leave the S flag set (=1), and the C, Z,

and V flags cleared (=0).

RET CC

INSTRUCTION FORMAT: (F9.4)

1 0 0 1 1 1 1 0 0 0 0 0 0 0

OPERATION:

 Nonsegmented
 Segmented

 if cc is true then
 if cc is true then

 PC <- @SP</td>
 PC <- @SP</td>

 SP <- SP + 2</td>
 SP <- SP + 4</td>

This instruction is used to return to a previously executing procedure at the end of a procedure entered by a CALL or CALR instruction. If the condition specified by "cc" is true, then the contents of the location addressed by the processor stack pointer are popped into the program counter (PC). The next instruction executed is that addressed by the new contents of the PC. See section 3.2.1 for a list of condition codes. The stack pointer used is R15 if nonsegmented, or RR14 if segmented. If the condition code is false, then control falls through to the instruction following the RET instruction.

FLAGS:

No flags affected

CYCLES:

Address
NS SS SL
10 -- 13 (return is taken)
7 -- 7 (return not taken)

EXAMPLE:

In nonsegmented mode, if the program counter contains \$2550, the stack pointer (R15) contains \$3000, location \$3000 contains \$1004, and the Z flag is clear, then the statement

RET NZ

will leave the value %3002 in the stack pointer and the program counter will contain %1004 (the address of the next instruction to be executed).

Rotate Left

RI. dst.src RLB

dst: R src: IM

INSTRUCTION FORMAT: (F1.1)

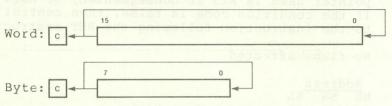
mode dst W/B: mode 1 1 0 0 1 W/B dst 0 0 S 0

(Perform operation once if s=0, twice if s=1)

OPERATION:

C <- dst(msb) $dst(0) \leftarrow dst(msb)$ $dst(n+1) \leftarrow dst(n)$ $\{n \text{ is } 0 \text{ to } msb-1\}$

The contents of the destination operand are rotated left one bit position if the source operand is 1, or two bit positions if the source operand is 2. The source operand may be omitted from the assembly language statement and thus defaults to the value 1. The most significant bit of the destination operand is moved to the bit O position and also replaces the C flag.



- FLAGS: C: Set if the last bit rotated from the most significant bit position was 1; cleared otherwise

 - Z: Set if the result is zero; cleared otherwise
 S: Set if the most significant bit of the result is set; cleared otherwise
 - V: Set if arithmetic overflow occurs, that is, if the sign of the destination changed during rotation; cleared otherwise
 - D: Unaffected

H: Unaffected

CYCLES:

Word/Byte (once) 6 7 (twice)

EXAMPLE: If register RH5 contains %88 (10001000), the statement

RLB RH5

will leave the value %11 (00010001) in RH5 and the carry flag will be set to one.

src

RLC dst,src

dst: R src: IM

INSTRUCTION FORMAT: (F1.1)

W/B: mode 1 1 0 0 1 W/B dst 1 0 s 0 10 R

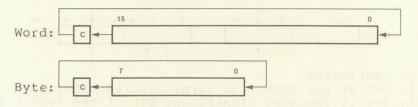
(Perform operation once if s=0, twice if s=1)

OPERATION:

dst(0) <- C C <- dst(msb)

 $dst(n+1) \leftarrow dst(n) \{n \text{ is } 0 \text{ to } msb-1\}$

The contents of the destination operand with the C flag are rotated left one bit position if the source operand is 1, or two bit positions if the source operand is 2. The source operand may be omitted from the assembly language statement and thus defaults to the value 1. The most significant bit of the destination operand replaces the C flag and the previous value of the C flag is moved to the bit 0 position of the destination.



FLAGS:

- C: Set if the last bit rotated from the most significant bit position was 1; cleared otherwise
- Z: Set if the result is zero; cleared otherwise
- S: Set if the most significant bit of the result is set; cleared otherwise
- V: Set if arithmetic overflow occurs, that is, if the sign of the destination changed during rotation; cleared otherwise
- D: Unaffected
- H: Unaffected

CYCLES:

Word/Byte
6 (once)
7 (twice)

EXAMPLE:

If the carry flag is clear (=0) and register R0 contains \$800F (1000000000001111), the statement

RLC RO, #2

will leave the value $\$003D\ (0000000000111101)$ in R0 and clear the carry flag.

RLDB dst, src

dst: R src: R

INSTRUCTION FORMAT: (F2.1)

B: mode 1 1 1 1 1 0 src dst

 $\frac{\text{mode}}{10}$ $\frac{\text{dst}}{R}$ $\frac{\text{src}}{R}$

OPERATION:

 $src(0:3) \leftarrow dst(0:3)$ $src(4:7) \leftarrow src(0:3)$ $dst(0:3) \leftarrow src(4:7)$

Rotates to the left BCD digits between source and destination byte operands. Simultaneously, the lower digit of destination is moved to the lower digit of source, the lower digit of source is moved to the upper digit of source, and the upper digit of source is moved to the lower digit of destination. The result is the destination after the operation, whose upper digit is unaffected. In multiple-digit BCD arithmetic, this instruction can be used to shift to the left a string of BCD digits, thus multiplying it by a power of ten.



FLAGS:

C: Unaffected

Z: Set if the result is zero; cleared otherwise

S: Set if the most significant bit of the result is set;

cleared otherwise

V: Unaffected D: Unaffected

H: Unaffected

CYCLES:

Byte

NOTE:

The same register must not be used in both the source and destination addressing mode designators.

RLDB Rotate Left Digit

EXAMPLE:

If location 100 contains the BCD digits 0,1 (00000001), location 101 contains 2,3 (00100011), and location 102 contains 4,5 (01000101)

100 101 102 4 5

the sequence of statements

!Set loop counter for 3 bytes (6 R3,#3 LD digits)! !Set pointer to low-order digits! T.D R2, #102 !Zero-fill low-order digit! CLRB RH1 LOOP: !Get next two digits! RL1,0R2 LDB !Shift digits left one position! RH1.RL1 RLDB !Replace shifted digits! T.DR @R2,RL1 !Advance pointer! DEC R2 !Repeat until counter is zero! R3.LOOP DJNZ

will leave the digits 1,2 (00010010) in location 100, the digits 3,4 (00110100) in location 101, and the digits 5,0 (01010000) in location 102.

100 101 102 5 0

Rotate Right

dst, srcope and and dst: Roomers src: IM RRB

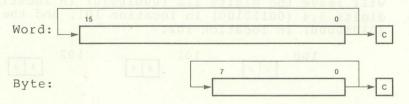
INSTRUCTION FORMAT: (F1.1)

mode dst src W/B: mode 1 W/B 10 1 1 0 0 0 s 0

(Perform operation once if s=0, twice if s=1)

OPERATION: C <- dst(0) dst(msb) <- dst(0) $dst(n) \leftarrow dst(n+1)$ {n is 0 to msb-1}

> The contents of the destination operand are rotated right one bit position if the source operand is 1, or two bit positions if the source operand is 2. The least significant bit of the destination operand is moved to the most significant bit and also replaces the C flag. The source operand may be omitted from the assembly language statement and thus defaults to the value 1.



FLAGS: C: Set if the last bit rotated from the least significant bit position was 1; cleared otherwise

Z: Set if the result is zero; cleared otherwise

S: Set if the most significant bit of the result is set; cleared otherwise

V: Set if arithmetic overflow occurs, that is, if the sign of the destination changed during rotation; cleared

otherwise D: Unaffected H: Unaffected

CYCLES:

Word/Byte 6 (once) 7 (twice)

EXAMPLE: If register RL6 contains %31 (00110001), the statement

> RRB RL6

will leave the value %98 (10011000) in RL6 and the carry flag will be set to one.

Rotate Right through Carry

RRC dst,src

dst: R src: IM

INSTRUCTION FORMAT: (F1.1)

W/B: mode 1 1 0 0 1 W/B dst 1 1 s 0

 $\frac{\text{mode}}{10}$ $\frac{\text{dst}}{\text{R}}$ $\frac{\text{src}}{\text{IM}}$

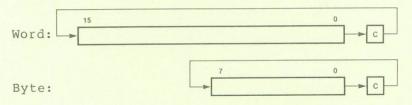
(Perform operation once if s=0, twice if s=1)

OPERATION:

dst(msb) <- C
C <- dst(0)</pre>

 $dst(n) \leftarrow dst(n+1)$ {n is 0 to msb-1}

The contents of the destination operand with the C flag are rotated right one bit position if the source operand is 1, or two bit positions if the source operand is 2. The least significant bit of the destination operand replaces the C flag and the previous value of the C flag is moved to the most significant bit position of the destination. The source operand may be omitted from the assembly language statement and thus defaults to the value 1.



FLAGS:

- C: Set if the last bit rotated from the least significant bit position was 1; cleared otherwise
 - Z: Set if the result is zero; cleared otherwise
 - S: Set if the most significant bit of the result is set; cleared otherwise
 - V: Set if arithmetic overflow occurs, that is, if the sign of the destination changed during rotation; cleared otherwise
 - D: Unaffected
 - H: Unaffected

CYCLES:

Word/Byte
6 (once)
7 (twice)

RRC

Rotate Right through Carry

EXAMPLE:

If the carry flag is clear (=0) and the register RO contains %00DD (0000000011011101), the statement

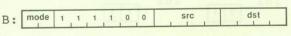
RRC RO,#2

will leave the value \$8037 (100000000110111) in RO and clear the carry flag.

RRDB dst,src dst: R

src: R

INSTRUCTION FORMAT: (F2.1)

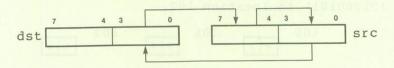


mode dst src 10

OPERATION:

src(4:7) <- dst(0:3) src(0:3) <- src(4:7) dst(0:3) < -src(0:3)

Rotates to the right BCD digits between source and destination byte operands. Simultaneously, the lower digit of destination is moved to the upper digit of source, the upper digit of source is moved to the lower digit of source, and the lower digit of source is moved to the lower digit of destination. The result is the destination after the operation, whose upper digit is unaffected. In multiple-digit BCD arithmetic, this instruction can be used to shift to the right a string of BDC digits, thus dividing it by a power of ten.



FLAGS: C: Unaffected

Z: Set if the result is zero; cleared otherwise

S: Set if the most significant bit of the result is set;

cleared otherwise

V: Unaffected D: Unaffected

H: Unaffected

CYCLES:

NOTE:

The same register must not be used in both the source and destination addressing mode designators.

RRDB Rotate Right Digit

EXAMPLE:

If location 100 contains the BCD digits 1,2 (00010010), location 101 contains 3,4 (00110100), and location 102 contains 5.6 (01010110)

100 101 102 5 6

the sequence of statements

LD R3,#3 !Set loop counter for 3 bytes (6 digits)!

LD R2,#100 !Set pointer to high-order digits! CLRB RH1 !Zero-fill high-order digit!

LOOP:

LDB RL1,@R2 !Get next two digits! RRDB RH1,RL1 !Shift digits right one position! LDB @R2,RL1 !Replace shifted digits! INC R2 !Advance pointer!

DJNZ R3,LOOP !Repeat until counter is zero!

will leave the digits 0,1 (00000001) in location 100, the digits 2,3 (00100011) in location 101, and the digits 4,5 (01000101) in location 102.

100 101 102 4 5

SBC dst.src SBCB

dst: R src: R

INSTRUCTION FORMAT: (F2.1)

W/B: mode 1 1 0 1 1 src

mode dst dst

OPERATION: mode 1 1 0 1 1 W/B

The source operand, along with the setting of the carry flag, is subtracted from the destination operand and the result is stored in the destination. The contents of the source are not affected. Subtraction is performed by adding the two's complement of the source operand to the destination operand. In multiple precision arithmetic, this instruction permits the carry ("borrow") from the subtraction of low-order operands to be subtracted from the subtraction of high-order operands.

- FLAGS: C: Cleared if there is a carry from the most significant bit of the result; set otherwise, indicating a "borrow"
 - Z: Set if the result is zero; cleared otherwise
 - S: Set if the result is negative; cleared otherwise
 - V: Set if arithmetic overflow occurs, that is, if the operands were of opposite signs and the sign of the result is the same as the sign of the source; cleared otherwise
 - D: SBC unaffected; SBCB set
 - H: SBC unaffected; SBCB cleared if there is a carry from the most significant bit of the low-order four bits of the result; set otherwise, indicating a "borrow"

CYCLES:

Word/Byte

EXAMPLE:

Long subtraction may be done with the following instruction sequence, assuming RO, Rl contain one operand and R2, R3 contain the other operand:

SUB R1,R3 !subtract low-order words! SBC RO, R2 !subtract carry and high-order words!

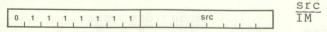
If RO contains %0038, Rl contains %4000, R2 contains %000A and R3 contains %F000, then the above two instructions leave the value %002D in RO and %5000 in Rl.

SC System Call

SC src

src: IM

INSTRUCTION FORMAT: (F9.5)



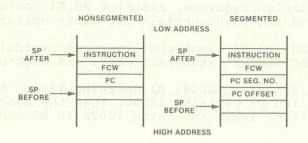
OPERATION:

| Nonsegmented | Segmented |
|----------------------|----------------------|
| SP <- SP - 4 | SP <- SP - 6 |
| @SP <- PS | @SP <- PS |
| SP <- SP - 2 | SP <- SP - 2 |
| @SP <- instruction | @SP <- instruction |
| PS <- System Call PS | PS <- System Call PS |
| | _ |

This instruction is used for controlled access to operating system software in a manner similar to a trap or interrupt. The current program status (PS) is pushed on the system processor stack, and then the instruction itself is pushed which includes the source operand (an 8-bit value). The PS includes the Flags and Control Word (FCW), and the updated program counter (PC). (The updated program counter value used is the address of the first instruction byte following the SC instruction.)

The system stack pointer is always used (R15 if nonsegmented, or RR14 if segmented), regardless of whether system or normal mode is in effect. The new PS is then loaded from the Program Status block associated with the System Call trap (see section 1.6.4), and control is passed to the procedure whose address is the program counter value contained in the new PS. This procedure may inspect the source operand on the top of the stack to determine the particular software service desired.

The following figure illustrates the format of the saved program status in the system stack:



FLAGS: No flags affected

CYCLES: Address

NS SS SL 33 -- 39

NOTES:

The Z8001 version always executes the segmented mode of the System Call instruction, regardless of the current mode, and sets the Segmentation Mode bit (SEG) to segmented mode (=1) at the start of the SC instruction execution. Both the Z8001 and Z8002 versions set the System/Normal Mode bit (S/N) to System mode (=1) at the start of the SC instruction execution. The status pins reflect the setting of these control bits during the execution of the SC instruction. However, the setting of SEG and S/N does not affect the value of these bits in the old FCW pushed on the stack. The new value of the FCW is not effective until the next instruction, so that the status pins will not be affected by the new control bits until after the SC instruction execution is completed.

The "src field" in the instruction format encoding contains the source operand. The "src field" values range from 0 to 255 corresponding to the source values 0 to 255.

EXAMPLE:

In the nonsegmented Z8002 version, if the contents of the program counter are %1000, the contents of the system stack pointer (R15) are %3006, and the program counter value associated with the System Call trap in the Program Status Area is \$2000, the statement

SC #3 !System call, request code=3!

causes the system stack pointer to be decremented to %3000. Location %3000 contains %7F03 (0111111100000011, the SC instruction). Location %3002 contains the old FCW, and location %3004 contains %1002 (the address of the instruction following the SC instruction). System mode is in effect, and the program counter contains the value %2000, which is the start of a System Call trap handler.

SDA dst,src SDAB SDAT.

dst: R src: R

INSTRUCTION FORMAT: (F6.6)

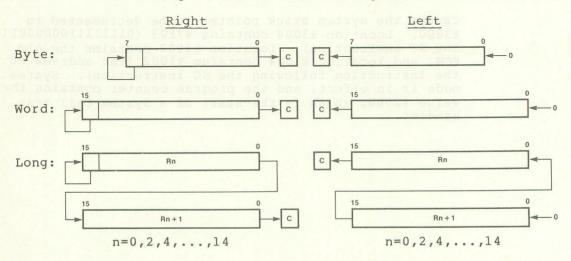
| W/B: | mo | de | 1 | 1 | 0 | 0 1 | W/B | 8 | ds | st | | 1 | 0 | 1 | 1 |
|------|----|----|---|---|----|-----|-----|-----|----|----|-----|---|-----|---|---|
| | 0 | 0 | 0 | 0 | se | src | se | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| L: | mo | de | 1 | 1 | 0 | 0 1 | 1 | IB. | d | st | No. | 1 | 1 | 1 | 1 |
| | | | - | - | | | - | | | | - | | 100 | - | - |

mode dst src 10 R

OPERATION: Right repeat src times: dst(msb) <- dst(msb)</pre> $dst(n) \leftarrow dst(n+1) \{ n \text{ is } 0 \text{ to } msb-1 \}$ C <- dst(0)

Left repeat src times: dst(0) <- 0 $dst(n+1) \leftarrow dst(n) \{n \text{ is } 0 \text{ to } msb-1\}$ baselomo C (- dst(msb) no separate of DR add notice

The destination operand is shifted arithmetically right or left the number of bit positions specified by the source operand (a word register). The shift count varies from -8 to +8 for SDAB, from -16 to +16 for SDA, and from -32 to +32 for SDAL, where a negative value is a right shift and a positive value is a left shift. A shift of zero positions does not affect the destination; however, the flags are set according to the destination value. The sign bit is replicated in shifts to the right, and the C flag is loaded from bit 0 of the destination. The least significant bit is filled with 0 in shifts to the left, and the C flag is loaded from the sign bit of the destination.



FLAGS: C: Set if the last bit shifted from the destination

was 1, cleared otherwise

Z: Set if the result is zero; cleared otherwise
S: Set if the result is negative; cleared otherwise

V: Set if arithmetic overflow occurs, that is, if the sign of the destination changed during shifting; cleared otherwise

D: Unaffected H: Unaffected

CYCLES: Word/Byte Long

15+3*n 15+3*n (n=number of bit positions, where 0 is equivalent to 1)

NOTE: The source operand is represented as a 16-bit two's

complement value. For each operand size, the operation is undefined if the value is not in the

range specified above.

EXAMPLE: If register R5 contains %C705 (1100011100000101) and

register Rl contains -2 (%FFFE or llllllllllllllll),

the statement

SDA R5, R1

performs an arithmetic right shift of two bit positions, leaves the value %FlCl (1111000111000001)

in R5, and clears the carry flag.

Shift Dynamic Logical

SDL dst,src dst: R src: R SDLL den harmalin engag el

INSTRUCTION FORMAT: (F6.6)

| W/B: | mode | | 1 1 | | 0 | 0 | 1 | W/B | dst | | | | 0 | 0 | 1 | 1 |
|------|------|----|-----|-----|-----|-----|---|-----|-----|---|----|---|---|---|---|---|
| | 0 0 | | 0 | 0 | src | | | | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| | mode | | | 1 1 | | 0 1 | | 1 | dst | | | 0 | 1 | 1 | 1 | |
| L: | mod | de | 1 | 1 | 0 | 0 | 1 | 1 | | d | st | | 0 | 1 | 1 | 1 |

mode dst src 10 R R

OPERATION: Right repeat src times:

dst(msb) <- 0

 $dst(n) \leftarrow dst(n+1) \{n \text{ is } 0 \text{ to } msb-1\}$

C <- dst(0)

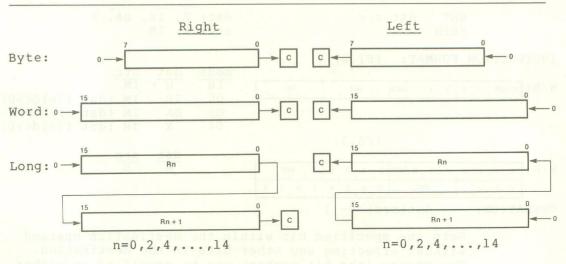
Left repeat src times:

dst(0) <- 0

 $d_{n} = d_{n} = d_{n$

C <- dst(msb)

The destination operand is shifted logically right or left the number of bit positions specified by the source operand (a word register). The shift count varies from -8 to +8 for SDLB, from -16 to +16 for SDL, and from -32to +32 for SDLL, where a negative value is a right shift and a positive value is a left shift. A shift of zero positions does not affect the destination; however, the flags are set according to the destination value. The most significant bit is filled with 0 in shifts to the right, and the C flag is loaded from bit 0 of the destination. The least significant bit is filled with 0 in shifts to the left, and the C flag is loaded from the most significant bit of the destination.



FLAGS: C: Set if the last bit shifted from the destination was 1; cleared otherwise

Z: Set if the result is zero; cleared otherwise

S: Set if the most significant bit of the result is set; cleared otherwise

V: Undefined
D: Unaffected
H: Unaffected

CYCLES: $\frac{\text{Word/Byte}}{15+3*n}$ $\frac{\text{Long}}{15+3*n}$ (n=number of bit positions, where 0 is equivalent to 1)

NOTE: The source operand is represented as a 16-bit two's complement value. For each operand size, the operation is undefined if the value is not in the range specified above.

EXAMPLE: If register RL5 contains %B3 (10110011) and register Rl contains 4 (000000000000000), the statement

SDLB RL5,Rl

performs a logical left shift of four bit positions, leaves the value %30 (00110000) in RL5, and sets the carry flag.

SET dst, src dst: R, IR, DA, X src: R. IM SETB INSTRUCTION FORMAT: (F1.2)mode dst src 10 R TM W/B: mode det 1 0 0 1 0 W/B cro 00 IR IM (dst field<>0) IM (dst field=0) 01 DA 0.1 TM (dst field<>0) X (F6.3)dst src R W/B: 0 1 0 W/B 0 0 0 1 0 0 0 0 src 0 0 0 0 0 0 0 dst(src) < -1OPERATION:

Sets the specified bit within the destination operand without affecting any other bits in the destination. The source (the bit number) can be specified as either an immediate value, or as a word register which contains the value. In the second case, the destination operand must be a register, and the source operand must be RO through R7 for SETB, or RO through R15 for SET. The bit number is a value from 0 to 7 for SETB or 0 to 15 for SET, with 0 indicating the least significant bit.

FLAGS:

No flags affected

Word/Byte CYCLES: dst src NS SS SL R IM 4 __ 11 IR IM 13 14 16 DA TM X IM 14 14 17 10 R

NOTE:

Only the lower four bits of the source operand are used to specify the bit number for SET, while only the lower three bits of the source operand are used with SETB. When the source operand is an immediate value, the "src field" in the instruction format encoding contains the bit number in the lowest four bits for SET, or the lowest three bits for SETB.

EXAMPLE:

If register RL3 contains %B2 (10110010) and register R2 contains the value 6, the statement

SETB RL3, R2

will leave the value %F2 (11110010) in RL3.

SETFLG flag

flag: C, Z, S, P, V

INSTRUCTION FORMAT: (F9.1)

1 0 0 0 1 1 0 1 C Z S P/V 0 0 0 1

OPERATION: FLAGS(4:7) <- FLAGS(4:7) OR instruction(4:7)

Any combination of the C, Z, S, P or V flags are set to one if the corresponding bit in the instruction is one. If the corresponding bit in the instruction is zero, the flag will not be affected. All other bits in the FLAGS register are unaffected. Note that the P and V flags are represented by the same bit. There may be one, two, three or four operands in the assembly language statement, in any order.

FLAGS: C: Set if specified; unaffected otherwise

Z: Set if specified; unaffected otherwise S: Set if specified; unaffected otherwise P/V: Set if specified; unaffected otherwise

D: Unaffected H: Unaffected

CYCLES: 7

EXAMPLE:

If the C, Z, and S flags are all clear (=0), and the P flag is set (=1), the statement

SETFLG C

will leave the C and P flags set (=1), and the Z and S flags cleared (=0).

SIN Special Input

SIN dst,src

dst: R src: DA

INSTRUCTION FORMAT: (F7.1)

W/B: 0 0 1 1 1 0 1 W/B dst 0 1 0 1

 $\frac{\text{dst}}{R}$ $\frac{\text{src}}{\text{DA}}$

OPERATION:

dst <- src

The contents of the source operand are loaded into the destination. I/O addresses are always 16 bits. This instruction is used to load information from the Memory Management Unit.

FLAGS:

No flags affected

CYCLES:

NOTES:

Word/Byte

This is a privileged instruction.

Special Input and Decrement

SIND dst,src,r SINDB dst: IR

INSTRUCTION FORMAT: (F6.4)

W/B: 0 0 1 1 1 0 1 W/B src 1 0 0 1 0 0 1 dst 1 0 0 0

 $\frac{\text{dst}}{\text{IR}} \quad \frac{\text{src}}{\text{IR}} \quad \frac{\text{r}}{\text{R}}$

OPERATION:

dst <- src
AUTODECREMENT dst {-1 if byte, -2 if word}
r <- r - 1</pre>

This instruction is used for block input of strings of data, typically status information from the Memory Management Unit. The contents of the I/O location addressed by the source register are loaded into the memory location addressed by the destination register. I/O addresses are always 16 bits. The destination register is then decremented by one if SINDB, or by two if SIND, thus moving the pointer to the previous element in the string. The word register specified by "r" (used as a counter) is then decremented by one.

FLAGS: C: Unaffected

Z: Undefined

S: Unaffected

V: Set if the result of decrementing r is zero; cleared otherwise

D: Unaffected H: Unaffected

CYCLES:

Word/Byte

NOTES:

This is a privileged instruction.

Special Input, Decrement and Repeat

SINDR dst,src,r

dst: IR src: IR

INSTRUCTION FORMAT: (F6.4)

 $\frac{\text{dst}}{\text{IR}} \quad \frac{\text{src}}{\text{IR}} \quad \frac{\text{r}}{\text{R}}$

OPERATION:

dst <- src AUTODECREMENT dst $\{-1 \text{ if byte, } -2 \text{ if word}\}$ r <- r - 1 repeat until r = 0

This instruction is used for block input of strings of data, typically status information from the Memory Management Unit. The contents of the I/O location addressed by the source register are loaded into the memory location addressed by the destination register. I/O addresses are always 16 bits. The destination register is then decremented by one if SINDRB, or by two if SINDR, thus moving the pointer to the previous element in the string. The word register specified by "r" (used as a counter) is then decremented by one. The entire operation is repeated until the result of decrementing r is zero. This instruction can input from 1 to 65536 bytes or 32768 words (the value for r must not be greater than 32768 for SINDR).

FLAGS:

C: Unaffected
Z: Undefined

S: Unaffected

V: Set

D: Unaffected H: Unaffected

CYCLES:

Word/Byte

11+10*n (n=number of data elements transferred)

NOTES:

This is a privileged instruction.

The status pins indicate a special I/O reference.

This instruction can be interrupted after each execution of the basic operation. The program counter value of the start of this instruction is saved before the interrupt request is accepted, so that the instruction can be properly resumed. Seven more cycles should be added to this instruction's execution time for each interrupt request that is accepted.

Special Input and Increment

SINI dst,src,r

dst: IR src: IR

INSTRUCTION FORMAT: (F6.4)

 $\frac{\text{dst}}{\text{IR}} \quad \frac{\text{src}}{\text{IR}} \quad \frac{\text{r}}{\text{R}}$

OPERATION:

dst <- src
AUTOINCREMENT dst {+1 if byte, +2 if word}
r <- r - 1</pre>

This instruction is used for block input of strings of data, typically status information from the Memory Management Unit. The contents of the I/O location addressed by the source register are loaded into the memory location addressed by the destination register. I/O addresses are always 16 bits. The destination is then incremented by one if SINIB, or by two if SINI, thus moving the pointer to the next element in the string. The word register specified by "r" (used as a counter) is then decremented by one.

FLAGS: C: Unaffected

Z: Undefined S: Unaffected

V: Set if the result of decrementing r is zero;

cleared otherwise

D: Unaffected H: Unaffected

CYCLES: Word/Byte

NOTES: This is a privileged instruction.

Special Input, Increment and Repeat

SINIR dst, src, r SINIRB

dst: IR src: IR

INSTRUCTION FORMAT: (F6.4)

1 W/B 0 0 0 0 1 1 1 dst 0 0 0 0

dst src IR

OPERATION:

dst <- src AUTOINCREMENT dst {+1 if byte, +2 if word} r <- r - 1 repeat until r = 0

This instruction is used for block input of strings of data, typically status information from the Memory Management Unit. The contents of the I/O location addressed by the source register are loaded into the memory location addressed by the destination register. I/O addresses are always 16 bits. The destination register is then incremented by one if SINIRB, or by two if SINIR, thus moving the pointer to the next element in the string. The word register specified by "r" (used as a counter) is then decremented by one. The entire operation is repeated until the result of decrementing r is zero. This instruction can input from 1 to 65536 bytes or 32768 words (the value for r must not be greater than 32768 for SINIR).

FLAGS:

C: Unaffected Z: Undefined S: Unaffected V: Set

D: Unaffected H: Unaffected

CYCLES:

Word/Byte 11+10*n (n=number of data elements transferred)

NOTES:

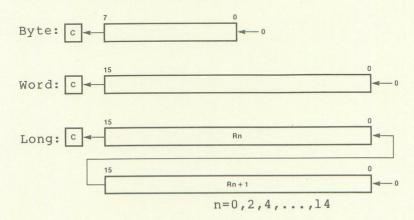
This is a privileged instruction.

The status pins indicate a special I/O reference.

This instruction can be interrupted after each execution of the basic operation. The program counter value of the start of this instruction is saved before the interrupt request is accepted, so that the instruction can be properly resumed. Seven more cycles should be added to this instruction's execution time for each interrupt request that is accepted.

SLA dst, src dst: R src: IM SLAB STAL (F5.1)INSTRUCTION FORMAT: mode dst src 10 R W/B: src L: mode repeat src times: OPERATION: dst(0) <- 0 $dst(n+1) \leftarrow dst(n)$ {n is 0 to msb-1} C <- dst(msb)

The destination operand is shifted arithmetically left the number of bit positions specified by the source operand. For SLAB, the source is in the range 0 to 8; for SLA, the source is in the range 0 to 16; for SLAL, the source is in the range 0 to 32. A shift of zero positions does not affect the destination; however, the flags are set according to the destination value. The least significant bit of the destination is filled with 0, and the C flag is loaded from the sign bit of the destination. This instruction performs a signed multiplication of the destination by a power of two with overflow indication. The source operand may be omitted from the assembly language statement and thus defaults to the value 1.



Shift Left Arithmetic

FLAGS: C: Set if the last bit shifted from the destination was 1: cleared otherwise

Z: Set if the result is zero: cleared otherwise

S: Set if the result is negative; cleared otherwise
V: Set if arithmetic overflow occurs, that is, if the sign of the destination changed during shifting; cleared

otherwise

D: Unaffected H: Unaffected

CYCLES: Word/Byte

Long 13+3*n

13+3*n (n=number of bit positions, where 0 is equivalent to 1)

The "src field" is encoded in the instruction format as NOTE .

the 16-bit two's complement positive value of the source operand. For each operand size, the operation is undefined if the source operand is not in the range specified above.

EXAMPLE: If register pair RR2 contains %1234ABCD, the statement

SLAL RR2,#8

will leave the value %34ABCD00 in RR2 and clear the carry flag.

220

SLL dst, src dst: R SLLB SLLL

src: IM

INSTRUCTION FORMAT: (F5.1)

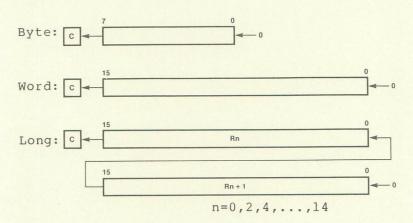
| W/B: | mode | 1 | 1 | 0 | 0 | 1 | W/B | dst | 0 | 0 | 0 | 1 |
|------|------|----|---|----|---|---|-----|--------|------|---|---|---|
| | | | | | | | src | | | | | |
| L: | mode | 1 | 1 | 0 | 0 | 1 | 1 | dst | 0 | 1 | 0 | 1 |
| | | no | | 80 | 9 | | src | o redm | Unt= | | | 1 |

mode dst src 10 R IM

OPERATION:

repeat src times: dst(0) <- 0 $dst(n+1) \leftarrow dst(n)$ {n is 0 to msb-1} C <- dst(msb)

The destination operand is shifted logically left the number of bit positions specified by the source operand. For SLLB, the source is in the range 0 to 8; for SLL, the source is in the range 0 to 16; for SLLL, the source is in the range 0 to 32. A shift of zero positions does not affect the destination; however, the flags are set according to the destination value. The least significant bit of the destination is filled with 0, and the C flag is loaded from the most significant bit of the destination. This instruction performs an unsigned multiplication of the destination by a power of two. The source operand may be omitted from the assembly language statement and thus defaults to the value 1.



FLAGS: C: Set if the last bit shifted from the destination was 1;

cleared otherwise

Z: Set if the result is zero; cleared otherwise

S: Set if the most significant bit of the result is set;

cleared otherwise

V: Undefined D: Unaffected

H: Unaffected

Word/Byte Long CYCLES:

13+3*n (n=number of bit positons, 13+3*n where 0 is equivalent to 1)

The "src field" is encoded in the instruction format NOTE: as the 16-bit two's complement positive value of the

source operand. For each operand size, the operation is undefined if the source operand is not in the

range specified above.

If register R3 contains %4321 (0100001100100001), EXAMPLE:

the statement

SLL R3, #1

will leave the value %8642 (1000011001000010) in R3 and clear the carry flag.

Special Output, Decrement and Repeat

SOTDR dst,src,r

dst: IR

INSTRUCTION FORMAT: (F6.4)

ISTRUCTION FORMAT: (10.

dst src r IR IR R

OPERATION:

dst <- src AUTODECREMENT src {-1 if byte, -2 if word} r <- r - 1 repeat until r = 0

This instruction is used for block output of strings of data, typically information to the Memory Management Unit. The contents of the memory location addressed by the source register are loaded into the I/O location addressed by the destination register. I/O addresses are always 16 bits. The source register is then decremented by one if SOTDRB, or by two if SOTDR, thus moving the pointer to the previous element in the string. The word register specified by "r" (used as a counter) is then decremented by one. The entire operation is repeated until the result of decrementing r is zero. This instruction can output from 1 to 65536 bytes or 32768 words (the value for r must not be greater than 32768 for SOTDR).

FLAGS: C:

C: Unaffected

Z: Undefined

S: Unaffected

V: Set

D: Unaffected

H: Unaffected

CYCLES:

Word/Byte

11+10*n (n=number of data elements transferred)

NOTES:

This is a privileged instruction.

The status pins indicate a special I/O reference.

This instruction can be interrupted after each execution of the basic operation. The program counter value of the start of this instruction is saved before the interrupt request is accepted, so that the instruction can be properly resumed. Seven more cycles should be added to this instruction's execution time for each interrupt request that is accepted.

Special Output, Increment and Repeat

SOTIR dst,src,r

dst: IR

INSTRUCTION FORMAT: (F6.4)

 $\frac{\text{dst}}{\text{IR}}$ $\frac{\text{src}}{\text{IR}}$ $\frac{\text{r}}{\text{R}}$

OPERATION:

dst <- src

AUTOINCREMENT src {+1 if byte, +2 if word}

r < -r - 1repeat until r = 0

This instruction is used for block output of strings of data, typically information to the Memory Management Unit. The contents of the memory location addressed by the source register are loaded into the I/O location addressed by the destination register. I/O addresses are always 16 bits. The source register is then incremented by one if SOTIRB, or by two if SOTIR, thus moving the pointer to the next element in the string. The word register specified by "r" (used as a counter) is then decremented by one. The entire operation is repeated until the result of decrementing r is zero. This instruction can output from 1 to 65536 bytes or 32768 words (the value for r must not be greater than 32768 for SOTIR).

FLAGS:

C: Unaffected

Z: Undefined

S: Unaffected

V: Set

D: Unaffected

H: Unaffected

CYCLES:

Word/Byte

11+10*n (n=number of data elements transferred)

NOTES:

This is a privileged instruction.

The status pins indicate a special I/O reference.

This instruction can be interrupted after each execution of the basic operation. The program counter value of the start of this instruction is saved before the interrupt request is accepted, so that the instruction can be properly resumed. Seven more cycles should be added to this instruction's execution time for each interrupt request that is accepted.

SOUT dst,src

dst: DA src: R

77 01

INSTRUCTION FORMAT: (F7.2)

W/B: 0 0 1 1 1 0 1 W/B Src 0 1 1 1

 $\frac{\text{dst}}{\text{DA}} \frac{\text{src}}{\text{R}}$

OPERATION:

dst <- src

The contents of the source operand are loaded into the destination. I/O addresses are always 16 bits. This instruction is used to load information into the Memory Management Unit.

FLAGS:

No flags affected

CYCLES:

Word/Byte par not isnusab and vd bazzanob

12

NOTES:

This is a privileged instruction.

SOUTD

Special Output and Decrement

SOUTD dst,src,r

dst: IR src: IR

INSTRUCTION FORMAT: (F6.4)

W/B: 0 0 1 1 1 0 1 W/B src 1 0 1 1 1 0 0 0 0

 $\frac{\text{dst}}{\text{IR}} \quad \frac{\text{src}}{\text{IR}} \quad \frac{\text{r}}{\text{R}}$

OPERATION:

dst <- src

AUTODECREMENT src $\{-1 \text{ if byte, } -2 \text{ if word}\}$ r $\langle -r-1 \rangle$

This instruction is used for block output of strings of data, typically information to the Memory Management Unit. The contents of the memory location addressed by the source register are loaded into the I/O location addressed by the destination register. I/O addresses are always 16 bits. The source register is then decremented by one if SOUTDB, or by two if SOUTD, thus moving the pointer to the previous element in the string. The word register specified by "r" (used as a counter) is then decremented by one.

FLAGS: C: Ur

C: Unaffected
Z: Undefined
S: Unaffected

V: Set if the result of decrementing r is zero;

cleared otherwise

D: Unaffected H: Unaffected

CYCLES:

Word/Byte

NOTES:

This is a privileged instruction.

Special Output and Increment

SOUTI dst,src,r

dst: IR

INSTRUCTION FORMAT: (F6.4)

dst src r

OPERATION:

dst <- src
AUTOINCREMENT src {+1 if byte, +2 if word}
r <- r -]</pre>

This instruction is used for block output of strings of data, typically information to the Memory Management Unit. The contents of the memory location addressed by the source register are loaded into the I/O location addressed by the destination register. I/O addresses are always 16 bits. The source register is then incremented by one if SOUTIB, or by two if SOUTI, thus moving the pointer to the next element in the string. The word register specified by "r" (used as a counter) is then decremented by one.

FLAGS:

C: Unaffected

Z: Undefined

S: Unaffected

V: Set if the result of decrementing r is zero;

cleared otherwise

D: Unaffected H: Unaffected

CYCLES:

Word/Byte

NOTES:

This is a privileged instruction.

Shift Right Arithmetic

SRA dst,src dst: R src: IM src: IM

INSTRUCTION FORMAT: (F5.1)

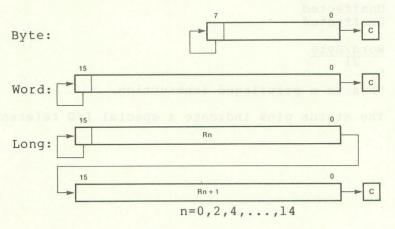
W/B: mode 1 1 0 0 1 W/B dst 1 0 0 1

L: mode 1 1 0 0 1 1 dst 1 1 0 1

src

OPERITION

The destination operand is shifted arithmetically right the number of bit positions specified by the source operand. For SRAB, the source is in the range 0 to 8; for SRA, the source is in the range 0 to 16; for SRAL, the source is in the range 0 to 32. A shift of zero positions does not affect the destination, however, the flags are set according to the destination value. The sign bit of the destination is replicated, and the C flag is loaded from bit 0 of the destination. The source operand may be omitted from the assembly language statement and thus defaults to the value 1.



C: Set if the last bit shifted from the destination FLAGS:

was 1; cleared otherwise

Z: Set if the result is zero; cleared otherwise S: Set if the result is negative; cleared otherwise

V: Cleared

D: Unaffected H: Unaffected

CYCLES:

Word/Byte 13+3*n

13+3*n

(n=number of bit positions, where 0 is equivalent to 1)

NOTE:

The "src field" is encoded in the instruction format as the 16-bit two's complement negative value of the source operand. For each operand size, the operation is undefined if the source operand is not in the range specified above.

EXAMPLE: If register RH6 contains %3B (00111011), the statement

SRAB RH6, #2

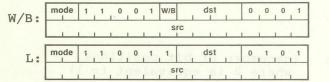
will leave the value %OE (00001110) in RH6 and set the carry flag.

SRL

Shift Right Logical

SRL dst,src dst: R
SRLB src: IM
SRLL

INSTRUCTION FORMAT: (F5.1)

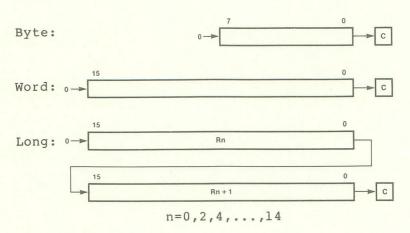


 $\frac{\text{mode}}{10}$ $\frac{\text{dst}}{\text{R}}$ $\frac{\text{src}}{\text{IM}}$

OPERATION:

ON: repeat src times:
 dst(msb) <- 0
 dst(n) <- dst(n+1) {n is 0 to msb-1}
 C <- dst(0)</pre>

The destination operand is shifted logically right the number of bit positions specified by the source operand. For SRLB, the source operand is in the range 0 to 8; for SRL, the source is in the range 0 to 16; for SRLL, the source is in the range 0 to 32. A shift of zero positions does not affect the destination, however, the flags are set according to the destination value. The most significant bit of the destination is filled with 0, and the C flag is loaded from bit 0 of the destination. This instruction performs an unsigned division of the destination by a power of two. The source operand may be omitted from the assembly language statement and thus defaults to the value of 1.



FLAGS:. C: Set if the last bit shifted from the destination was 1; cleared otherwise

Z: Set if the result is zero; cleared otherwise

S: Set if the most significant bit of the result is set; cleared otherwise

V: Undefined

D: Unaffected

H: Unaffected

CYCLES: Word/Byte 13+3*n

Long 13+3*n

(n=number of bit positions, where 0 is equivalent to 1)

NOTE:

The "src field" is encoded in the instruction format as the 16-bit two's complement negative value of the source operand. For each operand size, the operation is undefined if the source operand is not in the range specified above.

EXAMPLE:

If register R0 contains %llll (0001000100010001), the statement

SRL RO,#6

will leave the value \$0044 (0000000001000100) in RO and clear the carry flag.

SUB Subtract

SUB dst,src

SUBB SUBL dst: R

src: R, IM, IR, DA, X

INSTRUCTION FORMAT: (F2.1)

| T, mode 0 1 0 0 1 0 src | dst |
|-------------------------|-----|

dst src R mode R 10 IM (src field=0) 00 R IR (src field<>0) 00 R 01 R DA (src field=0) X (src field<>0) 01 R

OPERATION:

dst <- dst - src

The source operand is subtracted from the destination operand and the result is stored in the destination. The contents of the source are not affected. Subtraction is performed by adding the two's complement of the source operand to the destination operand.

FLAGS:

- C: Cleared if there is a carry from the most significant bit; set otherwise, indicating a "borrow"
- Z: Set if the result is zero; cleared otherwise
- S: Set if the result is negative; cleared otherwise
- V: Set if arithmetic overflow occurs, that is, if the operands were of opposite signs and the sign of the result is the same as the sign of the source; cleared otherwise
- D: SUB, SUBL unaffected; SUBB set
- H: SUB, SUBL unaffected; SUBB cleared if there is a carry from the most significant bit of the low-order four bits of the result; set otherwise, indicating a "borrow"

CYCLES:

| src | Wo | rd/B | yte | | Long | |
|-----|----|------|-----|----|------|----|
| | NS | SS | SL | NS | SS | SL |
| R | 4 | | | 8 | | |
| IM | 7 | | | 14 | | |
| IR | 7 | | | 14 | | |
| DA | 9 | 10 | 12 | 15 | 16 | 18 |
| X | 10 | 10 | 13 | 16 | 16 | 19 |

EXAMPLE:

If register RO contains %0344, the statement

SUB RO, #%AA

will leave the value %029A in RO.

TCC cc,dst

dst: R

INSTRUCTION FORMAT: (Fl.4)

W/B: mode 1 0 1 1 1 W/B dst cc

mode dst

OPERATION:

if cc is true then dst(0) <- 1

This instruction is used to create a Boolean data value based on the condition code resulting from a previous operation. The condition code specified by "cc" is tested. If the condition is true, then the least significant bit of the destination is set; otherwise it is unaffected. See section 3.2.1 for a list of condition codes. All other bits in the destination are unaffected.

unaffected

FLAGS: No flags affected

CYCLES: Word/Byte

EXAMPLE: If register Rl contains 0, and the Z flag is set, the

statement

TCC EQ, R1

will leave the value 1 in Rl.

TEST dst TESTB TESTL

dst: R, IR, DA, X

INSTRUCTION FORMAT: (F1.1)

| | | | | | | | 71 | U | | | 10 | mode | dst | | |
|------|------|----|---|---|---|----|-----|--------|--------|------|-----|------|-----|------|-----------|
| W/B: | mode | 0 | 0 | 1 | 1 | 0 | W/B | dst | 0 1 | 0 | 0 | 10 | R | | |
| | | | _ | _ | _ | _ | | 1 | o role | 1 | | 00 | IR | | |
| L: | mode | 0 | 1 | 1 | 1 | 0 | 0 | dst | 1 0 | 0 | 0 | 01 | DA | (dst | field=0) |
| | 7000 | 51 | | | | VI | | PIRBLE | 01 | 1916 | W B | 01 | X | (dst | field<>0) |

OPERATION: dst OR 0

The destination operand is tested (logically ORed with zero), and the appropriate flags set accordingly, which may then be used for arithmetic and logical conditional jumps. The contents of the destination are not affected.

FLAGS: C: Unaffected

Z: Set if the result is zero; cleared otherwise S: Set if the most significant bit of the result is set; cleared otherwise

P: TEST - unaffected; TESTL - undefined; TESTB - set if parity of the result is even; cleared otherwise

D: Unaffected H: Unaffected

CYCLES: Word/Byte dst Long SS SL NS SS SL R 7 -__ 13 8 13 IR DA 11 12 14 16 17 19 12 12 15 17 17 20

EXAMPLE: If register R5 contains %FFFF (11111111111111), the statement

> TEST R5

will set the S flag, clear the Z flag, and leave the other flags unaffected.

TRDB dst,src,r dst: IR

src: IR

INSTRUCTION FORMAT: (F6.4)

1 0 0 0 0 0 0 0 0 0 0 erc 0 0.

dst src r TR TR

OPERATION:

dst <- src[dst] AUTODECREMENT dst {-1} r (- r - 1

This instruction is used to translate a string of bytes from one code to another code. The contents of the location addressed by the destination register (the "target byte") is used as an index into a table of translation values whose lowest address is contained in the source register. The index is computed by adding the target byte to the address contained in the source register. The addition is performed following the rules for addressing mode arithmetic, with the target byte treated as an unsigned 8-bit value extended with high-order zeros. The sum is used as the address of an 8-bit translation value within the table which replaces the original contents of the location addressed by the destination register.

The destination register is then decremented by one, thus moving the the pointer to the previous element in the string. The word register specified by "r" (used as a counter) is then decremented by one.

Because the 8-bit target byte is added to the source register to obtain the address of a translation value, the table may contain 256 bytes. A smaller table size may be used where it is known that not all possible 8-bit target byte values will occur.

FLAGS:

C: Unaffected

7: Undefined

S: Unaffected

V: Set if the result of decrementing r is zero; cleared

otherwise D: Unaffected

H: Unaffected

CYCLES:

Byte

TRDB

Translate and Decrement

NOTE:

The original contents of register RHl is lost, and is replaced by an undefined value.

EXAMPLE:

If register R6 contains %4001, the byte at location %4001 contains 3, register R9 contains %1000, the byte at location %1003 contains %AA, and register R12 contains 2, the statement

TRDB @R6, @R9, R12

will leave the value %AA in location %4001, the value \$4000 in R6, and the value 1 in R12. R9 will not be affected.

Translate, Decrement and Repeat

TRDRB dst.src.r

dst: IR src: TR

INSTRUCTION FORMAT: (F6.4)

| B: | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | dst | 1 | 1 | 0 | 0 |
|----|---|---|---|---|---|----|---|-----|-----|---|---|---|---|
| | 0 | 0 | 0 | 0 | 9 | ne | r | 5 0 | src | 0 | 0 | 0 | 0 |

dst src r TR R IR IR

OPERATION: dst <- src[dst] AUTODECREMENT dst {-1} r < -r - 1repeat until r = 0

> This instruction is used to translate a string of bytes from one code to another code. The contents of the location addressed by the destination register (the "target byte") is used as an index into a table of translation values whose lowest address is contained in the source register. The index is computed by adding the target byte to the address contained in the source register. The addition is performed following the rules for addressing mode arithmetic, with the target byte treated as an unsigned 8-bit value extended with high-order zeros. The sum is used as the address of an 8-bit translation value within the table which replaces the original contents of the location addressed by the destination register.

The destination register is then decremented by one, thus moving the pointer to the previous element in the string. The word register specified by "r" (used as a counter) is then decremented by one. The entire operation is repeated until the result of decrementing r is zero. This instruction can translate from 1 to 65536 bytes.

Because the 8-bit target byte is added to the source register to obtain the address of a translation value, the table may contain 256 bytes. A smaller table size may be used where it is known that not all possible 8-bit target byte values will occur.

FLAGS:

C: Unaffected

7: Undefined

S: Unaffected

V: Set

D: Unaffected

H: Unaffected

Translate, Decrement and Repeat

CYCLES .

Byte

11+14*n (n=number of data elements translated)

NOTES .

The original contents of register RHl is lost, and is replaced by an undefined value.

This instruction can be interrupted after each execution of the basic operation. The program counter of the start of this instruction is saved before the interrupt request is accepted, so that the instruction can be properly resumed. Seven more cycles should be added to this instruction's execution time for each interrupt request that is accepted.

EXAMPLE: If register R6 contains %4002, the bytes at locations %4000 through %4002 contain the values %00, %40, %80, respectively, register R9 contains %1000, the translation table from location %1000 through %10FF contains 0,1,2,...,%7F,0,1,2,...,%7F (the second zero is located at %1080), and register R12 contains 3, the statement

TRDRB @R6, @R9, R12

will leave the values %00, %40, %00 in locations %4000 through %4002, respectively. Register R6 will contain %3FFF, and R12 will contain 0. R9 will not be affected.

| | BEFORE | | _Brancad |
|-------|-----------------|---------|----------|
| | and the same | % 1000 | 00000000 |
| %4000 | 00000000 | %1001 | 00000001 |
| %4001 | 0 1 0 0 0 0 0 0 | %1002 | 00000010 |
| %4002 | 10000000 | | • |
| | | | |
| | | • | • |
| | AFTER | %107F | 01111111 |
| | to saethi | %1080 | 00000000 |
| %4000 | 00000000 | %1081 | 00000001 |
| %4001 | 01000000 | % 1082 | 00000010 |
| %4002 | 00000000 | 890 .57 | 9370 39 |
| | | • | |
| | | • | |
| | | %10FF | 01111111 |

Translate and Increment

TRIB dst, src, r la data dst: IR

aw benilebau src: IR enal

INSTRUCTION FORMAT: (F6.4) no beau at not tour tent at

dst 0 0 0 0

dst src r O O O O IR IR R

OPERATION: dst <- src[dst] AUTOINCREMENT dst {+1} r (- r - 1

> This instruction is used to translate a string of bytes from one code to another code. The contents of the location addressed by the destination register (the "target byte") is used as an index into a table of translation values whose lowest address is contained in the source register. The index is computed by adding the target byte to the address contained in the source register. The addition is performed following the rules for addressing mode arithmetic, with the target byte treated as an unsigned 8-bit value extended with high-order zeros. The sum is used as the address of an 8-bit translation value within the table which replaces the original contents of the location addressed by the destination register.

The destination register is then incremented by one, thus moving the pointer to the next element in the string. The word register specified by "r" (used as a counter) is then decremented by one.

Because the 8-bit target byte is added to the source register to obtain the address of a translation value, the table may contain 256 bytes. A smaller table size may be used where it is known that not all possible 8-bit target byte values will occur.

FLAGS: C: Unaffected

Z: Undefined S: Unaffected

V: Set if the result of decrementing r is zero; cleared

otherwise D: Unaffected H: Unaffected

Byte CYCLES:

Translate and Increment

NOTE .

The original contents of register RHl is lost, and is replaced by an undefined value.

EXAMPLE:

This instruction is used in a "loop" of instructions which translate a string of data from one code to any other desired code, but an intermediate operation on each data element is required. The following sequence translates a string of 1000 bytes to the same string of bytes, with all ASCII "control characters" (values less than 32, see Appendix) translated to the "blank" character (value=32). A test, however, is made for the special character "return" (value=13) which terminates the loop. The translation table contains 256 bytes. The first 33 (0-32) entries all contain the value 32, and all other entries contain their own index in the table, counting from zero.

| | LD LD LD | R3,#1000 R1,#STRING R2,#TABLE | !Initialize counter! !Load start addresses! |
|------|----------------|-------------------------------------|--|
| LOOP | : dide8 | | |
| | CPB | @R1,#13 | !Check for return character! |
| | JR | EQ, DONE | !Exit loop if found! |
| | TRIB | @R1, @R2, R3 | !Translate next byte! |
| | JR | NOV, LOOP | !Repeat until counter = 0! |
| DONE | | | |

| TABLE + 0 | 00100000 |
|-------------|-----------|
| TABLE + 1 | 00100000 |
| TABLE + 2 | 00100000 |
| 9. 93 | |
| 10. 288 | ibbs. edi |
| | d oct nin |
| TABLE + 32 | 00100000 |
| TABLE + 33 | 00100001 |
| TABLE + 34 | 00100010 |
| • | • |
| • | • |
| • | • |
| TABLE + 255 | 11111111 |
| | |

240

ulsy bearieban src: IR

INSTRUCTION FORMAT: (F6.4)

src 0 0 0 0 0 0 0 0

 $\begin{array}{ccc} \text{dst} & \text{src} & \text{r} \\ \overline{\text{IR}} & \overline{\text{IR}} & \overline{\text{R}} \end{array}$

OPERATION:

dst <- src[dst] AUTOINCREMENT dst {+1} r <- r - 1 repeat until r = 0

This instruction is used to translate a string of bytes from one code to another code. The contents of the location addressed by the destination register (the "target byte") is used as an index into a table of translation values whose lowest address is contained in the source register. The index is computed by adding the target byte to the address contained in the source register. The addition is performed following the rules for addressing mode arithmetic, with the target byte treated as an unsigned 8-bit value extended with highorder zeros. The sum is used as the address of an 8-bit translation value within the table which replaces the original contents of the location addressed by the destination register.

The destination register is then incremented by one, thus moving the pointer to the next element in the string. The word register specified by "r" (used as a counter) is then decremented by one. The entire operation is repeated until the result of decrementing r is zero. This instruction can translate from 1 to 65536 bytes.

Because the 8-bit target byte is added to the source register to obtain the address of a translation value, the table may contain 256 bytes. A smaller table size may be used where it is known that not all possible 8-bit target byte values will occur.

FLAGS: C: Unaffected

Z: Undefined

S: Unaffected

V: Set

D: Unaffected

H: Unaffected

CYCLES:

11+14*n (n=number of data elements translated)

TRIRB

Translate, Increment and Repeat

NOTES .

The original contents of register RHI is lost, and is replaced by an undefined value.

This instruction can be interrupted after each execution of the basic operation. The program counter of the start of this instruction is saved before the interrupt request is accepted, so that the instruction can be properly resumed. Seven more cycles should be added to this instruction's execution time for each interrupt request that is accepted.

EXAMPLE:

The following sequence of instructions can be used to translate a string of 80 bytes from one code to another. The pointers to the string and the translation table are set, the number of bytes to translate is set, and then the translation is accomplished.

and mi beniednos LD seedbookl. #STRING of seulay dollar R2, #TABLE R2, #TABLE R3, #80 TRIRB ONL ORL, OR2, R3 DO DOMESTICAL DESCRIPTION OF THE PROPERTY OF THE PROPER for addressing mode arithmetic, with the target byte treated as an unsigned 8-bit value extended with high

Translate, Test and Decrement

TRTDB srcl, src2, r

srcl: IR

INSTRUCTION FORMAT: (F6.4)

B: 1 0 1 1 1 0 0 0 src1 1 0 1 0 0 0 src2 0 0 0 0 0

 $\frac{\text{srcl}}{\text{IR}} = \frac{\text{src2}}{\text{IR}} = \frac{\text{r}}{\text{R}}$

OPERATION:

RH1 <- src2[src1]
AUTODECREMENT src1 {-1}
r <- r - 1

This instruction is used to scan a string of bytes testing for bytes with special meaning. The contents of the location addressed by the first source register (the "target byte") is used as an index into a table of translation values whose lowest address is contained in the second source register. The index is computed by adding the target byte to the address contained in the second source register. The addition is performed following the rules for addressing mode arithmetic, with the target byte treated as an unsigned 8-bit value extended with high-order zeros. The sum is used as the address of an 8-bit value within the table which is loaded into register RH1. The Z flag is set if the value loaded into RHl is zero; otherwise the Z flag is cleared. The contents of the locations addressed by the source registers are not affected.

The first source register is then decremented by one, thus moving the pointer to the previous element in the string. The word register specified by "r" (used as a counter) is then decremented by one.

Target byte values which have corresponding zero translation table entry values are to be scanned over, while target byte values which have corresponding non-zero translation table entry values are to be detected. Because the 8-bit target byte is added to the second source register to obtain the address of a translation value, the table may contain 256 bytes. A smaller table size may be used where it is known that not all possible 8-bit target byte values will occur.

TRTDR

Translate, Test and Decrement

FLAGS. C: Unaffected

Z: Set if the translation value loaded into RHl is zero; cleared otherwise

S: Unaffected

V: Set if the result of decrementing r is zero; cleared otherwise

D: Unaffected H: Unaffected

CYCLES.

Byte 25

EXAMPLE: If register R6 contains %4001, the byte at location %4001 contains 3, register R9 contains %1000, the byte at location %1003 contains %AA, and register R12 contains 2, the statement

TRTDB @R6, @R9, R12

will leave the value %AA in RHl, the value %4000 in R6, and the value 1 in R12. Location %4001 and register R9 will not be affected.

TRTDRB srcl, src2, r

r srcl: IR src2: IR

INSTRUCTION FORMAT: (F6.4)

| : | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | src1 | 1 | 1 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|------|---|---|---|---|
| | 0 | 0 | 0 | 0 | | | r | | src2 | 1 | 1 | 1 | 0 |

 $\frac{\text{srcl}}{\text{IR}} \quad \frac{\text{src2}}{\text{IR}} \quad \frac{\text{r}}{\text{R}}$

OPERATION:

RH1 <- src2[src1]
AUTODECREMENT src1 {-1}
r <- r - 1
repeat until RH1 <> 0 or r = 0

This instruction is used to scan a string of bytes testing for bytes with special meaning. The contents of the location addressed by the first source register (the "target byte") is used as an index into a table of translation values whose lowest address is contained in the second source register. The index is computed by adding the target byte to the address contained in the second source register. The addition is performed following the rules for addressing mode arithmetic, with the target byte treated as an unsigned 8-bit value extended with high-order zeros. The sum is used as the address of an 8-bit value within the table which is loaded into register RH1. The Z flag is set if the value loaded into RHl is zero; otherwise the Z flag is cleared. The contents of the locations addressed by the source registers are not affected.

The first source register is then decremented by one, thus moving the pointer to the previous element in the string. The word register specified by "r" (used as a counter) is then decremented by one. The entire operation is repeated until either the Z flag is clear, indicating that a non-zero translation value was loaded into RH1, or until the result of decrementing r is zero. This instruction can translate and test from 1 to 65536 bytes.

Target byte values which have corresponding zero translation table entry values are to be scanned over, while target byte values which have corresponding non-zero translation table entry values are to be detected. Because the 8-bit target byte is added to the second source register to obtain the address of a translation value, the table may contain 256 bytes. A smaller table size may be used where it is known that not all possible 8-bit target byte values will occur.

TRTDRR

Translate, Test, Decrement and Repeat

FLAGS: C: Unaffected

Z: Set if the translation value loaded into RHl is zero: cleared otherwise

S: Unaffected

V: Set if the result of decrementing r is zero; cleared otherwise

D: Unaffected H: Unaffected

CYCLES:

Byte 11+14*n (n=number of data elements tested)

NOTE:

This instruction can be interrupted after each execution of the basic operation. The program counter of the start of this instruction is saved before the interrupt request is accepted, so that the instruction can be properly resumed. Seven more cycles should be added to this instruction's execution time for each

interrupt request that is accepted.

EXAMPLE: If register R6 contains %4002, the bytes at locations %4000 through %4002 contain the values %00, %40, %80, respectively, register R9 contains %1000, the translation table from location %1000 through %10FF contains 0,1,2,...,%7F,0,1,2,...,%7F (the second zero is located at %1080), and register R12 contains 3, the statement

TRTDRB @R6, @R9, R12

will leave the value %40 in RHl (which was loaded from location %1040). Register R6 will contain %4000, and R12 will contain 1. R9 will not be affected.

| %4000 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
|-------|---|---|---|---|---|---|---|---|
| %4001 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| %4002 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

| %1000 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
|-------|---|---|---|---|---|---|----|---|
| %1001 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| %1002 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |
| | - | | * | | | 7 | Y. | 1 |
| | | | | | • | | | |
| 11 | 9 | | | | | | | |
| %107F | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| %1080 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| %1081 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| %1082 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |
| 9.1 | 9 | 0 | 1 | U | 0 | 8 | | Ь |
| | 9 | | | 6 | 0 | | | |
| | 8 | | | | 0 | | | |
| %10FF | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| | _ | - | - | - | - | - | - | - |

Translate, Test and Increment

TRTIB srcl, src2, r srcl: IR IHE offi bebool suley not talen src2: IR - 448. IN INSTRUCTION FORMAT: (F6.4) 0 0 0

1 0 0 1 1 0 0 0 0 Src2 0 0 0 0

srcl src2 r IR IR R

OPERATION:

RH1 <- src2[src1] AUTOINCREMENT srcl {+1} r <- r - 1

This instruction is used to scan a string of bytes testing for bytes with special meaning. The contents of the location addressed by the first source register (the "target byte") is used as an index into a table of translation values whose lowest address is contained in the second source register. The index is computed by adding the target byte to the address contained in the second source register. The addition is performed following the rules for addressing mode arithmetic, with the target byte treated as an unsigned 8-bit value extended with high-order zeros. The sum is used as the address of an 8-bit value within the table which is loaded into register RHl. The Z flag is set if the value loaded into RHl is zero; otherwise the Z flag is cleared. The contents of the locations addressed by the source registers are not affected.

The first source register is then incremented by one, thus moving the pointer to the next element in the string. The word register specified by "r" (used as a counter) is then decremented by one.

Target byte values which have corresponding zero translation table entry values are to be scanned over, while target byte values which have corresponding non-zero translation table entry values are to be detected. Because the 8-bit target byte is added to the second source register to obtain the address of a translation value, the table may contain 256 bytes. A smaller table size may be used where it is known that not all possible 8-bit target byte values will occur.

Translate, Test and Increment

FLAGS: C: Ilnaffected

> Z: Set if the translation value loaded into RHl is zero; cleared otherwise

S: Unaffected

V: Set if the result of decrementing r is zero; cleared otherwise

D: Unaffected H: Unaffected

CYCLES:

EXAMPLE: This instruction is used in a "loop" of instructions which translate and test a string of data, but an intermediate operation on each data element is required. The following sequence outputs a string of 72 bytes, with each byte of the original string translated from its 7-bit ASCII code (see Appendix) to an 8-bit value with odd parity. A test, however, is made for the special character "return" (value=%D) which terminates the loop. The translation table contains 128 bytes, which assumes that the most significant bit of each byte in the string is always zero. Each entry contains its own index in the table in the low-order seven bits, with the most significant bit of the value set or cleared to give odd parity for the entire value.

R4.#72 !Initialize counter! R5, #STRING !Load start addresses! LD LD R6, #TABLE

LOOP:

CPB @R5, #%D !Check for return character! JR EQ,DONE !Exit loop if found!
TRTIB @R5,@R6,R4 !Translate next byte! OUTB PORTN, RHl !Output with odd parity! JR NOV, LOOP ! Repeat until counter=0!

DONE:

| TABLE + 0 | 10000000 | added to the sea |
|-------------------------------|----------|------------------|
| TABLE +1 | 00000001 | address of a tr |
| beau ed vem esta el TABLE+2 | 00000010 | tain 256 bytes. |
| past did-8 sldrazog [TABLE+3] | 10000011 | where it is kno |
| TABLE + 4 | 00000100 | byte values wil |
| TABLE + 5 | 10000101 | |
| TABLE + 6 | 10000110 | |
| TABLE + 7 | 00000111 | |
| TABLE + 8 | 00001000 | |
| TABLE + 9 | 10001001 | |
| • | • | |
| | • | |

TRTIRB srcl, src2, r

srcl: IR

INSTRUCTION FORMAT: (F6.4)

| B: | 1 | 0 | 1 | . 1 | 1 | 0 | 0 | 0 | src1 | 0 | 1 | 1 | 0 |
|----|---|---|---|-----|---|---|---|---|------|---|---|---|---|
| | 0 | 0 | 0 | 0 | | | r | | src2 | 1 | 1 | 1 | 0 |

 $\frac{\text{srcl}}{\text{IR}} \quad \frac{\text{src2}}{\text{IR}} \quad \frac{\text{r}}{\text{R}}$

OPERATION:

RH1 <- src2[src1]
AUTOINCREMENT src1 {+1}
r <- r - 1
repeat until RH1 <> 0 or r = 0

This instruction is used to scan a string of bytes testing for bytes with special meaning. The contents of the location addressed by the first source register (the "target byte") is used as an index into a table of translation values whose lowest address is contained in the second source register. The index is computed by adding the target byte to the address contained in the second source register. The addition is performed following the rules for addressing mode arithmetic, with the target byte treated as an unsigned 8-bit value extended with high-order zeros. The sum is used as the address of an 8-bit value within the table which is loaded into register RHI. The Z flag is set if the value loaded into RHI is zero; otherwise the Z flag is cleared. The contents of the locations addressed by the source registers are not affected.

The first source register is then incremented by one, thus moving the pointer to the next element in the string. The word register specified by "r" (used as a counter) is then decremented by one. The entire operation is repeated until either the Z flag is clear, indicating that a non-zero translation value was loaded into RHI, or until the result of decrementing r is zero. This instruction can translate and test from 1 to 65536 bytes.

Target byte values which have corresponding zero translation table entry values are to be scanned over, while target byte values which have corresponding non-zero translation table entry values are to be detected. Because the 8-bit target byte is added to the second source register to obtain the address of a translation value, the table may contain 256 bytes. A smaller table size may be used where it is known that not all possible 8-bit target byte values will occur.

Translate, Test, Increment and Repeat

FLAGS: C: Ilnaffected

> Z: Set if the translation value loaded into RHl is zero; cleared otherwise

S: Ilnaffected

V: Set if the result of decrementing r is zero; cleared

otherwise D: Unaffected

H: Unaffected

CYCLES: Byte

11+14*n (n=number of data elements tested)

This instruction can be interrupted after each execution of the basic operation. The program counter of the start of this instruction is saved before the interrupt request is accepted, so that the instruction can be properly resumed. Seven more cycles should be added to this instruction's execution time for each interrupt request that is accepted.

EXAMPLE: The following sequence of instructions can be used to scan a string of 80 bytes, testing for special characters as defined by corresponding non-zero translation table entry values. The pointers to the string and translation table are set, the number of bytes to scan is set, and then the translation and testing is accomplished. The Z and V flags can be tested after the operation to determine if a special character was found, and whether the end of the string has been reached. The translation value loaded into RH1 might then be used to index another table, or to select one of a set of sequences of instructions to execute next.

LD R4, #STRING EL DELL S and LD die | R5, #TABLE ger el molderago clear, indicating t08#, 68 con-zer diranslation value TRTIRB 0 @R4, @R5, R6 on beback as

aad boa adalanad o JR oo dow NZ, SPECIAL . 0198 at 1 pot

END OF STRING:

19VO ben SPECIAL: 1 918 gentav vidne eldat noidalanaat

pathogosation JR data OV, LAST_CHAR_SPECIAL non-zero translation table entry values are to be

detected. Because the 8-bit target byte is added to the second source register to obtain the address of

Target byte values which have corresponding zero

LAST_CHAR_SPECIAL: delicated the control of the con

TSETB

TSET dst dst: R, IR, DA, X

INSTRUCTION FORMAT: (Fl.1)

| | | | | | | | | | CHIC | mode | dst | aT. | |
|-----------|-----|-----|----|---|-----|--------|-----|---|------|------|-----|------|-----------|
| W/B: mode | 0 (| 1 | 1 | 0 | W/B | dst | 0 1 | 1 | 0 | 10 | R | H. | |
| W/ 2 | HUL | MAI | ac | V | 0 0 | eriter | 00 | | | 00 | IR | | |
| | | | | | | | | | | 01 | DA | (dst | field=0) |
| | | | | | | | | | | 01 | X | (dst | field<>0) |

OPERATION .

S (- dst(msb) dst(0:msb) <- 111...111

Tests the most significant bit of the destination operand, copying its value into the S flag, then sets the entire destination to all 1 bits. This instruction provides a locking mechanism which can be used to synchronize software "processes" which require exclusive access to certain data or instructions at one time.

- FLAGS: C: Unaffected
 - 7: Unaffected
 - S: Set if the most significant bit of the destination was 1: cleared otherwise
 - V: Unaffected
 - D: Unaffected H: Unaffected

CYCLES:

| dst | Word/Byte | | | | | | |
|--|-----------|----|----|--|--|--|--|
| Acceptant Control of C | NS | SS | SL | | | | |
| R | 7 | | | | | | |
| IR | 11 | | | | | | |
| DA | 14 | 15 | 17 | | | | |
| X | 15 | 15 | 18 | | | | |

NOTE:

During the execution of this instruction, BUSRQ is not honored in the time between loading the destination from memory and storing the destination to memory. For systems with one processor, this ensures that the testing and setting of the destination will be completed without any intervening accesses. This instruction should not be used to synchronize software "processes" residing on separate processors where the destination is a shared memory location, since this locking mechanism cannot be guaranteed to function correctly with multi-processor accesses.

TSFT Test and Set

EXAMPLE .

A simple mutually-exclusive "critical region" may be implemented by the following sequence of statements:

ENTER:

TSET SEMAPHORE

.00

JR MI, ENTER !loop until resource controlled by SEMAPHORE

is available!

!Critical Region--only one software process executes this code at a time!

CLR SEMAPHORE !release resource controlled by SEMAPHORE! XOR dst,src

dst: R src: R, IM, IR, DA, X

INSTRUCTION FORMAT: (F2.1)

| | | | | | * | | | | |
|------|------|---|---|---|---|---|-----|-----|-----|
| W/B: | mode | 0 | 0 | 1 | 0 | 0 | W/B | src | dst |
| | | | | | | | | | |

| node | dst | src | XXX | |
|------|-----|-----|------|-----------|
| 10 | R | R | DFXX | |
| 00 | R | | | field=0) |
| 00 | R | IR | (src | field<>0) |
| 01 | R | | | field=0) |
| 01 | R | X | (src | field<>0) |

OPERATION: dst <- dst XOR src

The source operand is logically EXCLUSIVE ORed with the destination operand and the result is stored in the destination. The contents of the source are not affected. The EXCLUSIVE OR operation results in a one bit being stored whenever the corresponding bits in the two operands are different; otherwise, a zero bit is stored.

FLAGS: C: Unaffected

Z: Set if the result is zero; cleared otherwise
S: Set if the most significant bit of the result is set; cleared otherwise

P: XOR - unaffected; XORB - set if parity of the result is even; cleared otherwise

D: Unaffected H: Unaffected

CYCLES:

| src | Word/Byte | | | | | | |
|-----|-----------|----|----|--|--|--|--|
| | NS | SS | SL | | | | |
| R | 4 | | | | | | |
| IM | 7 | | | | | | |
| IR | 7 | | | | | | |
| DA | 9 | 10 | 12 | | | | |
| X | 10 | 10 | 13 | | | | |

EXAMPLE:

If register RL3 contains %C3 (11000011) and the source operand is the immediate value %7B (01111011), the statement

XORB RL3, #%7B

will leave the value %B8 (10111000) in RL3.

3.4 UNIMPLEMENTED INSTRUCTIONS

As explained in Section 1.6.4, certain binary instruction codes are defined by the Z8000 architecture, but are not currently implemented by the hardware. The following opcodes (listed by the first instruction word in hexadecimal where "x" means any value) will cause an Unimplemented Instruction trap if an attempt is made to execute them:

0Exx 0Fxx 4Exx 4Fxx 8Exx 8Fxx

NOTE

These opcodes are reserved for future extensions to the Z8000 architecture and must not be used by the programmer. To provide for compatibility with Zilog hardware and software systems, the results of attempted execution of these instructions is undefined, other than the occurrence of an Unimplemented Instruction trap.

Section 4 Structuring a Z8000 Program

4.1 INTRODUCTION

This section introduces the high-level PLZ/ASM statements described in Section 5. The structuring of programs and the concepts of module linkage and relocation are discussed.

4.2 PROGRAM STRUCTURE

4.2.1 Modules

A Z8000 PLZ/ASM program consists of one or more separately-coded and assembled $\frac{\text{modules.}}{\text{modules}}$ These modules are combined into an executable program using the module linkage and relocation facilities of an operating system such as Z-80 RIO. One of the modules should include a "main program"; that is, a GLOBAL procedure whose name is supplied to the linking facility as the program's entry point.

PLZ/ASM modules are made up of high-level and assembly-language statements that either declare or define data or perform some action. The assembly-language statements described in Section 3 are action statements. In general, data definition equates an identifier with a fixed value or type, whereas data declaration equates an identifier with a variable and associates a type with it.

Data, labels, or procedures can be shared between modules by declaring them to be GLOBAL in one module and declaring them as EXTERNAL in other modules that reference them. Data, labels, or procedures declared as INTERNAL to a module may be referenced only within that module. See Section 4.2.5 for more explanation of the scope of these objects.

The following example is the skeleton of a module showing two procedures, the second of which is a main program.

```
bubble sort MODULE
```

CONSTANT

true := 1

false ·= 0 !Constant definition!

EXTERNAL.

list ARRAY [10 WORD] !External variable

declared in another module

TNTERNAL. sort A SIG Tovel Hotel

PROCEDURE TO BUILDING SHOULD BE TO ALLOS IN DECL LOSS

!First procedure! !Procedure name!

!Second procedure!

END sort

!End first procedure!

GLOBAL main

PROCEDURE

LD R0, #9*2 STUDON SASAN LESTONOM DOLLMANDE LO CALL SORT

END bubble sort

END main !End second procedure!

!End of module!

4.2.2 Procedures

A procedure declaration defines an executable portion of a module (including both action and data statements). It also associates an identifier with that block of code so that it can be activated by the assembly-language CALL statement. (See the "CALL sort" statement in the preceding example).

Every procedure declaration also has a scope associated with it. It can be:

- GLOBAL (the procedure can be called from other modules);
- INTERNAL (it can be called only within the current module);
- EXTERNAL (it is called from the current module, but is declared as GLOBAL in another module).

A procedure declaration can include LOCAL variables or label declarations as shown in the following example. LOCAL variables and labels can be referenced only within the procedure in which they are declared.

INTERNAL !Procedure scope! sort !Procedure name! PROCEDURE LOCAL switch BYTE !Local variable! ENTRY DO !DO loop introduced! ! (action statements)! OD !End DO loop! !End of procedure! END sort

Note in this example that the keyword ENTRY is used to separate the declarations of LOCAL variables from the executable action statements making up the body of the procedure. The example also carries the previous "bubble-sort module" example to the first inner level of detail--the procedure. At the same time it introduces another program structuring element--the DO loop.

4.2.3 DO Loops

The description of the DJNZ instruction in Section 3 pointed out that it is used primarily for loop control. DO loops perform a similar function, but in a more structured, high-level manner. The statements between the keywords DO and OD are executed repeatedly until control is diverted through a loop control statement: REPEAT, EXIT, or one of the assembly-language branch statements. The EXIT statement causes execution to continue at the first statement following the innermost DO loop containing the EXIT, while REPEAT statement causes execution to continue at the first statement of the innermost DO loop containing the REPEAT.

A DO statement does not introduce a new scope. It contains only action statements.

The following example enlarges upon the original "bubblesort" skeleton and also introduces the IF statement.

DO

LDB switch, #false
CLR Rl

DO !Note nested DO loop!

CP Rl, RO
IF UGE THEN EXIT FI !IF statement at inner level!

...
INC Rl, #2
OD !End nested DO loop!

CPB switch, #false
IF . . . FI !IF statement at outer level!
!End DO loop!

Note in this example that DO loops can contain IF statements or other DO loops. IF statements can also have DO loops or other IF statements nested within them.

4.2.4 IF Statements

The general form of the IF statement is:

IF condition-code THEN actionl ELSE action2 FI

The IF statement specifies that all statements between THEN and ELSE (action1) be executed if the condition specified is true. Otherwise, the statements between ELSE and FI (action2) are executed. The ELSE clause can be omitted, in which case no action is performed when the specified condition is false. Like the DO loop, the IF statement contains only executable statements.

The condition codes specified are the same as for the assembly language JP and JR instructions (section 3.2.1). As in the case of DO loops versus the DJNZ instruction, one can set up conditional statements using only assembly-language instructions, but the high-level IF statement may lead to a more structured program.

The following example shows a brief IF statement followed by a longer one.

in the current module but declared as GLOBAL in another, module.

INTERNAL specifies that the variable is declared in the current

Base of IF UGE THEN EXIT FI page JAMETIXE . Manager and address

IF UGT THEN
LDB switch, #true
LD list(R1), R6
LD list(R2), R4

4.2.5 Scope

The scope of a variable, label, or procedure, refers to that portion of the program in which the object is recognized and handled in keeping with its declaration. When declaring an object for use solely within the current module or procedure, its use elsewhere is of no concern. Even though the same identifier is used in another scope, it refers to a different object.

In general, a scope can be an entire executable program, a module, or a procedure. An identifier is accessible in a scope S if it is either

- Declared in S or in the module containing

 S, or
- Declared EXTERNAL in the module containing S. Managara

New identifiers are declared by their appearance in a variable, procedure, constant, or type declaration, or by their use as a label within the module. If a label identifier is not declared explicitly, it is assumed to have an INTERNAL scope and thus is accessible throughout the module in which it is defined. The scope of a label identifier may explicitly be declared to be either GLOBAL, INTERNAL, or LOCAL. In addition, a special form of statement labels is discussed in Section 5.3.5, which always have a LOCAL scope and therefore cannot be declared explicitly.

New identifiers introduced in a variable, procedure, or label declaration are accessible only within the newly-established scope as determined from the declaration class specified: GLOBAL, EXTERNAL, INTERNAL or LOCAL. A new identifier cannot be the same as an identifier already accessible in the current scope.

For <u>variable</u> and <u>label declarations</u>, GLOBAL specifies that the variable is declared in the current module but may be used in other modules. GLOBAL variables have a scope of the entire executable program. EXTERNAL specifies that the variable is used in the current module but declared as GLOBAL in another module. INTERNAL specifies that the variable is declared in the current module and is accessible only within the current module. LOCAL specifies that the variable can be accessed only inside the procedure in which it is declared. GLOBAL, EXTERNAL, and INTERNAL can only be specified at the module level; LOCAL can only be used at the procedure level.

For variables declared with type RECORD, the scope of a record field is the module or procedure in which it is defined. Note that this implies that record fields must have names that are unique within their scope.

Procedure declarations must be either GLOBAL, EXTERNAL, or INTERNAL, as defined for variables above. The name given to the procedure as part of the procedure declaration is recognized in the entire scope of the enclosing module.

Constant identifiers are defined using the CONSTANT class, and can be defined only at the module level. The scope of a constant identifier is the module scope, and it cannot be used outside the module unless it is redefined.

Type identifiers are defined using the TYPE class. They can be defined only at the module level and are recognized only within the scope of the module.

An identifier used in a scope and not declared in that scope is said to be free in that scope. Any identifier that is free in the scope of a procedure must be declared in the enclosing module. Procedures do not explicitly import variables into their scope as modules do (via the EXTERNAL declaration).

4.2.6 Summary

The preceding sections were intended only as an introduction to program structuring. The detailed formats of the declarations used to define the program structure are listed in Section 5. Even without those details, however, the following module should be comprehensible at this point. If any statement looks foreign, reread the part of Section 3 or 4 that explains its operation.

An example of a complete bubble_sort module for sorting a 10-word array is as follows:

| bubble_sort MODULE | !Module declaration! |
|---|---|
| false := 0 true := 1 | !Constant declarations! |
| EXTERNAL list ARRAY [10 WORD] | |
| INTERNAL quieval partiews et alut | !Procedure declaration! |
| LOCAL switch BYTE ENTRY DO | !Local variable declaration! !Loop control switch! |
| LDB switch, #false CLR Rl DO | !Initialize switch! !Clear array pointer i! |
| CP R1, R0 IF UGE THEN EXIT FI | !Done?! |
| LD R2, R1 INC R2, #2 LD R4, list(R1) LD R6, list(R2) | !Initialize pointer j! !j = i+l (double for words)! |
| CP R4, R6 IF UGT THEN LDB switch, #true LD list(R1), R6 LD list(R2), R4 | <pre>!If list[i] > list[j] then! !exchange to bubble largest! !number to top of array!</pre> |
| FI INC R1, #2 | !Advance pointer (double for |
| OD CPB switch, #false IF EQ THEN RET FI | words)! !End nested DO loop! !Test switch! |
| OD END sort | !End outer DC loop! !End of procedure! |
| GLOBAL main PROCEDURE | !New procedure declaration! !Program entry procedure! |
| ENTRY LD R0, #9*2 | !No LOCALs! !Initialize loop control! !Double for word array! !Call sort procedure! |
| END main | !End of main procedure! |
| END bubble_sort | !End of module! |

4.3 RELOCATABILITY

The Z8000 PLZ/ASM assembler produces relocatable object modules. Essentially, this frees the programmer from memory management concerns during program development (since object code can be relocated in memory) and also allows programs to be developed in modules whose addresses are resolved automatically when the modules are linked.

Modular program development offers numerous advantages to the developer. Complex programs can be divided into several smaller tasks and assigned to a development team, should schedule constraints require quick completion. An entire program need not be delayed while one module is awaiting development, since modules can be separately tested and assembled. A change affecting a single module will not have a ripple effect through the entire program and will require reassembly of the affected module only. In short, several simple programs are generally easier to write, test, and debug than one large, complex program.

The relocatability feature of the Z8000 assembler is supported by the \$ABS, \$REL, and \$SECTION assembler directives. These directives determine whether programs are assembled in relocatable mode or not, and where data and action statements are to be loaded into memory. An operating system program called a linker relocates object modules and resolves inter-module references.

These assembler directives are summarized below. Other directives are listed in Appendix C, and are described in detail in the Z8000 Assembler User's Guide.

4.3.1 Sections

In addition to the logical structuring provided by modules and procedures, it is possible to divide a program into sections which can be mapped into various areas of memory when the program is linked or loaded for execution. For example, the programmer may choose to group a set of data structures and the procedures which manipulate them together in the same module. But it may also be desirable to physically separate the object code for the procedures from the data in a system where read-only memory is used for the procedures and read/write memory is used for the data.

The assembler allows a program to be arbitrarily divided into named sections. Each section might be allocated to a different address space; for instance, code or data memory. In the segmented Z8000, each section might be mapped into a different segment, or several sections might be combined from different modules into the same segment. A single module may contain

several sections, each of which will be allocated a different area in memory. Alternatively, the portions of a single section may be spread through several modules and the portions will be automatically combined into a single area by the linker. Sections provide the programmer with complete control over the mapping of a program into the address spaces of the Z8000 processor.

Usually the full generality of arbitrary sections is not needed for a particular application. By default, the assembler automatically creates two sections for each module, one for all the data variables and the other for all procedures. The data section is given the same name as the module name, with the suffix "D" (or "_d") appended. The procedure section is given the same name as the module name, with the suffix "P" (or "_p") appended. If the first character of the module name is uppercase, then an uppercase suffix is used; otherwise, a lowercase suffix is used.

For example, the data section "bubble_sort_d" will be created for the sample module in Section 4.2.6, and contains only the BYTE named "switch". The program section "bubble_sort_p" contains all the code in the module.

The \$SECTION assembler directive can be used to override the default sectioning. This directive causes all following statements to be associated with a symbolic identifier which can later be mapped into a particular memory area. It has the format:

\$SECTION identifier

The \$SECTION directive remains in effect until either another \$SECTION directive is encountered, the \$SDEFAULT directive is encountered which returns to the default section scheme described above, or the end of the module is found.

NOTE

Because the assembler automatically calculates the displacement used in the Relative addressing mode (Section 2.4.6), the target address must be in the same section and module as the instruction which uses Relative Addressing.

4.3.2 Location Counter Control

The assembler keeps track of the "location" of the current statement with a $\frac{1}{1}$ counter, just as an executing program does with its program counter. There is a location counter associated with each section in a program. The counter value

represents a 16-bit offset within the current section. The offset may be either an absolute value, or it may represent a relocatable value which may be adjusted depending on where the module's portion of the section is finally allocated at link or load time. Relocatable and absolute portions of a section may be specified in the same module or program.

If the \$ABS assembler directive has been specified, the location counter reflects the absolute location of the current statement; if \$REL has been specified, the counter reflects the relocatable offset of the statement. If neither is specified, the counter defaults to relocatable offset 0 at the beginning of a module. The location counter symbol "\$" may be used in any expression, and represents the address of the first byte of the current instruction.

For the majority of programming tasks, one need not specify either \$ABS or \$REL in the source program. Program location can be carried out more appropriately at link time or load time. If one desires complete control over program location at assembly time, however, the \$ABS directive will force that part of the program to reside at a specific location.

4.3.3 Modes of Arithmetic Expressions

All arithmetic expressions have one of three modes associated with them: absolute, relocatable, or external.

An absolute expression consists of one or more constants, constant identifiers, or absolute labels combined with arithmetic or logical operators. The difference between two relocatable expressions is also considered to be absolute.

JP IRQ VECTOR

!Where IRQ VECTOR is an absolute label!

ADD R1, #K*3

!Where K is a constant identifier!

A relocatable expression is one containing exactly one identifier subject to relocation after assembly. The expression can be extended by adding or subtracting an absolute expression. Plus and minus are the only operators allowed, however.

JP Z, LOOP+2

!Where LOOP is a relocatable label!

An <u>external</u> expression is one containing exactly one external identifier, possibly extended by adding or subtracting an absolute expression. An external identifier is one that is used in the current module but defined in another module (Section 4.2.5). The value of an external identifier is not known until the modules are linked.

LD RO, timer_count !External if timer_count was defined outside current module!

In the following summary, "AB" stands for an absolute value or expression, "RE" stands for a relocatable value or expression, and "EX" is an external identifier or expression. "operator" is one of the standard arithmetic or logical operators (+, -, *, LOR, LAND, etc.).

"AB" is defined as either:

- an absolute identifier
- AB "operator" AB
- +AB, -AB, or LNOT AB
- RE RE

"RE" is defined as either:

- a relocatable identifier
- RE + AB
- AB + RE
- RE AB
- +RE

"EX" is defined as either:

- an external identifier
- EX + AB
- AB + EX
- EX AB
- +EX

Certain mode combinations are not permitted in PLZ/ASM. For example, a relocatable expression that does not result in simple relocation is invalid. Simple relocation means that only a single relocation factor need be added to a relocatable value when the assembled program is relocated.

In general, the second example would be invalid no matter which arithmetic or logical operator was used to combine LOOPl and LOOP2. The one exception is subtraction. Suppose LOOP2 is the label of the first statement following a procedure named LOOP1. The statement

R5, #(LOOP2 - LOOP1) T.D

could be used to find the length of the LOOPl procedure. The difference between two relocatable labels (which must be in the same section) is always absolute, regardless of where the module is relocated to.

Other invalid mode combinations are:

• A relocatable expression multiplied or divided by an absolute expression:

> Z, LOOP1*4 !INVALID! asbi asploads as a JP

 A relocatable expression subtracted from an absolute expression (although the reverse is allowed):

> DJNZ R5, LOOP1 - 8 DJNZ R5. 8 - LOOP1 !INVALID!

• An external expression combined with a relocatable expression, and vice-versa:

> ADD R6, EXTERNAL NUMB + LOOP1 !INVALID! Z, LOOP1 - EXTERNAL NUMB !INVALID!

!VALID!

Section 5 PLZ/ASM High-Level Statements

5.1 78000 SOURCE PROGRAM STATEMENTS

The majority of code in a Z8000 PLZ/ASM program will normally be the assembly-language instructions described in Section 3. Typically, the source program will also include some high-level statements and assembler directives.

High-level statements perform two basic functions:

- They introduce program structures (modules, procedures, DO loops, and IF statements);
- They declare and define data.

Assembler directives control the mode of assembly (absolute or relocatable), determine where object code is to be stored in memory, and specify the form of assembler output. These directives are embedded in the source program and are always preceded by a dollar sign (\$) (see Appendix C).

In the descriptions of high-level statements in this section the following notational conventions are used:

- Keywords are shown as all capital letters: MODULE
- Parameters shown in lowercase letters represent items to be replaced by actual data or names: module identifier
- Optional items are enclosed in square brackets: [local declaration]
- Possible repetition of an item is indicated by appending a "+" (to signify one or more repetitions) or an "*" (to signify zero or more repetitions) to the item: declaration*
- Other special characters shown in statement and command formats such as :=, (), and [] will be enclosed in single quotes and must be written as shown. The special symbol ":=" means "is defined as" or "is assigned".

For example:

CONSTANT
 constant_identifier ':=' constant_expression
RECORD '[' identifier+ type ']'

5.2 PROGRAM STRUCTURING STATEMENTS

5.2.1 Module Declaration

A Z8000 PLZ/ASM program module consists of a sequence of data and procedure declarations. These declarations are bounded by the module declaration statement and the end-of-module statement. The format of a module declaration is:

module_identifier MODULE

declaration*
END module_identifier

where

module_identifier

conforms to the rules for identifiers (Section 2.2.1).

declaration

is zero or more data and/or procedure declarations (Sections 5.2.2 and 5.3).

Example:

sine_computation MODULE

END sine computation

5.2.2 Procedure Declaration

A procedure declaration defines an executable part of a program and associates an identifier with it so that it can be activated by the assembly-language CALL statement.

The procedure heading specifies the identifier naming the procedure. This identifier labels the first instruction in the procedure, and can be used as any other program label. The scope of the procedure identifier may be either GLOBAL, INTERNAL, or EXTERNAL. If the procedure is declared EXTERNAL, then only the procedure identifier is given, since the actual definition of the procedure occurs in some other module. A procedure declaration may also include local variable declarations. These variables

are recognized only within the procedure in which they are declared.

The format of the procedure declaration is:

procedure_identifier PROCEDURE
 [LOCAL
 [variable_identifier+ type]*]*
 [ENTRY
 action_statement*]
END procedure identifier

where

identifier conforms to the rules for identifiers (Section 2.2.1).

is BYTE, SHORT_INTEGER, INTEGER, WORD, LONG, LONG_INTEGER, LABEL, ARRAY, RECORD or a user_defined type (Sections 5.3.2 through 5.3.4).

action_statement is zero or more assembly-language,
DO, IF, REPEAT, or EXIT statements,
or a combination of these
statements.

The keyword ENTRY is used to separate the local variable declarations from the executable part of the procedure, and must be used whenever any action statements are specified.

Example:

Print PROCEDURE Discourse Discourse

GLOBAL suml BYTE

add_routine PROCEDURE
LOCAL
sum BYTE
ENTRY
LDB RH2,suml
ADDB RH2,#72
LDB sum,RH2
CALL print
RET
END add routine

Note that the RET instruction precedes the END statement. If RET is not present, control will fall through to the statement following the END statement.

5.2.3 DO Statement Sample of the middle wife besinpober

DO loops provide a framework for performing actions repetitively. The statements between the DO and OD keywords are executed repeatedly until control is diverted through a loop control statement.

The only way the execution flow of a DO loop can be diverted is by encountering an assembly-language branch instruction (DJNZ, JP, JR, CALL, CALR, RET, or IRET) or an EXIT or REPEAT statement (described below) .

The format of the DO loop is:

[label]* DO action_statement* ARRAY, RECORD OF a user de COed

where

label days conforms to the rules for labels addensed TIXE to Transpar (Sections 2.2.1 and 5.3.5) and is used to identify the DO block for use with multi-level EXIT and REPEAT statements.

action_statement is zero or more assembly-language, DO, IF, REPEAT, or EXIT statements, or a combination of these statements.

The assembler automatically inserts a single unconditional jump instruction at the OD keyword which branches back to the DO keyword. Either a JR or JP is generated depending on the range of the loop; JR is used whenever possible.

The EXIT statement causes execution to continue at the first statement following the innermost DO...OD block containing the EXIT. The EXIT statement may be further qualified by a label indicating a specific DO...OD block from which to exit. Its format is:

EXIT [FROM label]

where

label

conforms to the rules for labels (Sections 2.2.1 and 5.3.5).

The assembler automatically inserts a single unconditional jump to the instruction following the indicated OD keyword. Either a JR or JP is generated depending on the range of the EXIT, with a JR used whenever possible.

The REPEAT statement causes execution to continue at the first statement of the innermost DO...OD block containing the REPEAT. It can also be qualified by a label indicating a specific DO...OD block to which execution is to proceed. Its format is:

REPEAT [FROM label]

where

label

conforms to the rules for labels (Sections 2.2.1 and 5.3.5).

The assembler automatically inserts a single unconditional jump instruction at the REPEAT statement which branches to the indicated DO keyword. Either a JR or JP is generated depending on the range of the REPEAT, with a JR used whenever possible.

The assembler automatically inserts a single condition:

LOOP1: DO

ADD RO, @Rl

INC R5, #2

CP R5, #limitl

IF EQ THEN EXIT FI

DO ADD R2, @R3 INC R6, #2 CP R6, #limit2

IF GT THEN REPEAT FROM LOOP1 FI OD OD depending on the range of the IF state GO

5.2.4 IF Statement

The IF statement specifies that the statements between the keywords THEN and ELSE (or between THEN and FI if the ELSE clause is omitted) are to be executed if the specified condition code is If the condition is false and the ELSE clause is present, true. the statements between ELSE and FI are executed. If the condition is false and the ELSE clause is omitted, execution continues with the statement following FI.

The format of the IF statement is:

IF condition code THEN action statement* [ELSE action2 statement*] FI

where

condition code

is Z, NZ, C, NC, PL, MI, EQ, NE, OV, NOV, PE, PO, LE, LT, GT, GE, ULE, ULT, UGT, or UGE (Section 3.2.1).

action1 statement

is performed if "condition_code" is true and consists of zero or more assembly-language, DO, IF, REPEAT, or EXIT statements or a combination of these statements.

action2 statement

is performed if "condition_code" is false and consists of zero or more assembly-language, DO, IF, REPEAT, or EXIT statements or a combination of these statements.

The assembler automatically inserts a single conditional jump instruction just before the THEN keyword, which branches to either the ELSE clause, if present, or to the FI keyword if not. The conditional jump has an opposite sense from the condition code given; for instance, "IF OV THEN" generates a "JP NOV" instruction. The opposing condition pairs are: Z-NZ, C-NC, PL-MI, EQ-NE, OV-NOV, PE-PO, LE-GT, LT-GE, ULE-UGT, and ULT-UGE. If the ELSE clause is present, a single unconditional jump is inserted just before the ELSE clause which branches to the FI keyword.

For each of the jump instructions, either a JR or JP is generated depending on the range of the IF statement, with a JR used whenever possible.

Example:

IF NZ THEN
LD counter, #1
FI

IF GT
THEN SETFLG C; RET
ELSE RESFLG C; RET
FI

5.2.5 IF-CASE Statement

The IF-CASE statement is an extension of the IF statement. It allows the user to select from a series of actions depending on the contents of a selector register. The case whose list contains a match with the contents of the selector register is performed. An ELSE clause can be used to specify alternative statements to be executed if no match occurs. If no ELSE is specified and no match occurs, the statement following the FI keyword is executed next.

The IF-CASE statement has the format:

selector register [CASE expression+ THEN action statement*]+ [ELSE action statement*]

FT

where

selector_register is the designator for a single byte, word, or long word byte, word, or long word

register (Section 2.4.2).

expression is any expression which is valid as an operand in a Compare

instruction.

action_statement is zero or more assemblylanguage, DO, IF, REPEAT, or

EXIT statements or a combination

of these statements.

The assembler automatically inserts a Compare instruction (CP, CPB, or CPL) and a conditional jump for each list element. For the last expression in a CASE clause, a "jump NE" to the next CASE is generated (or to the ELSE clause, if present, or if not present, to the FI keyword for the last CASE). Either a JR or JP instruction is used depending on the range of the CASE clause, with a JR used whenever possible. When there is more than one expression in a single CASE clause, all but the last expression generate a "JR EQ" to the start of the action statements associated with the CASE clause. Therefore, the number and size of expressions for a single CASE clause must not exceed the range of the first JR instruction for that clause.

Example:

IF R5 CASE #1, R4 THEN CALL control 1 CASE #2, @R3 THEN CALL control gt 1 ELSE CALL control qt 4 FI

5.3 DEFINING DATA

Data (constants and variables) must be defined or declared so that it can be referenced accordingly. In general, data definition associates an identifier with a fixed value or type. Data declaration introduces an identifier as the name of a variable and associates a scope and type with it. The following three statements are used to define and declare data:

- The constant-definition statement (CONSTANT), which associates a constant identifier with a fixed value;
- The <u>type-definition</u> statement (TYPE), which associates a type identifier with a fixed type;
- The <u>variable declaration</u> which associates a variable identifier with a scope, type, and (optionally) an initial value.

Constant identifiers are assumed to have INTERNAL scope. Constants have no explicit type and are represented as 32-bit values.

Type identifiers appearing in type-definitions are also assumed to have INTERNAL scope. No scope or initial value can be specified in type-definition statements. They are used primarily to categorize data or provide a template for structured data.

The variable declaration allows a variable identifier to be associated with any specific scope, type, or initial value (within the limits of the variable's scope).

5.3.1 Constant Definition

A constant definition associates an identifier with a constant expression. Since this value must be determinable at assembly time, any identifier appearing in the expression must be previously defined.

Constant identifiers have no explicit type and are always represented as 32-bit values. And, since constants are defined at the module level, they have a scope of the module in which they are defined (INTERNAL). Constants to be used in other modules must be redefined in those modules.

The format of the constant-definition statement is:

CONSTANT

[constant identifier ':=' constant expression]*

where

constant identifier conforms to the rules for identifiers (Section 2.2.1).

constant expression conforms to the rules for constant expressions (Section 2.3.2).

Example: Sendy willness 11d-CE

CONSTANT

minus := -1 count := 10

A SYMB MODULUS := 256 and apappore fey Idmagas and set of the later than the later than the set of the later than the l

5.3.2 Data Types

Data types are associated with variables either to indicate the size of the values the variables can represent or to identify a name as a label. Simple data types indicate whether a variable can hold an 8-bit, 16-bit or 32-bit value; structured data types provide a template of storage for collections of simple variables. The label type is used to declare the scope of labels explicitly.

Data types can be directly associated with variable identifiers in variable declarations (Section 5.3.4), or they may be associated with variables indirectly using a user-defined "type identifier" to specify the type. The latter is an identifier that has been previously associated with a type in a TYPE statement (Section 5.3.3). In the following example, the variable CHAR is associated with the simple type BYTE, allowing CHAR to be used as a type identifier in subsequent type definitions: yam assubs a inemele is functioned A . sysw inemelle

TYPE

CHAR BYTE

letter CHAR

digit CHAR

Simple Types. The basic data type is either a standard simple data type or a simple data type defined by the user in a type definition (like CHAR in the example just given). The standard data types are:

SHORT INTEGER or BYTE An 8-bit quantity whose value can be signed (-128 to +127)or unsigned (0 to 255). This value may also represent a single character from the ASCII character set.

INTEGER or WORD A 16-bit quantity whose value can be signed (-32768 to +32767) or unsigned (0 to oed) anolesedate instanc65535).

LONG INTEGER or LONG

A 32-bit quantity whose value can be signed (-2,148,483,648 to +2,148,483,647) or unsigned (0 to 4,296,967,296).

The values of simple variables (that is, variables defined with simple types) are interpreted as signed or unsigned depending on their use in assembly-language instructions.

> TYPE CHAR BYTE small value BYTE large value LONG INTEGER letter CHAR

Structured Types. Structured types are defined by indicating the structuring method to be used and the types of all elements within the selected structure. Two structuring methods are available: ARRAY and RECORD.

Array Structures. An array structure is a collection of variable elements, each of which has the same type. When referenced, the identifier associated with the ARRAY type refers to the entire array structure. Arrays with N elements are indexed from 0 to N-1; for example, a 10-element array has index 0 as the first element and index 9 as the last element.

Individual elements within an array can be accessed in several different ways. A particular element's address may be calculated at run-time, for instance, by specifying an indexed address mode. At assembly-time, a particular element's address can be specified by an expression containing the array identifier and a fixed offset or by an array identifier followed by one or more constant expressions enclosed within square brackets. In the latter case, each constant expression represents an index for the particular "dimension" of the array, and the assembler's calculation of the desired element's address may involve an implicit multiplication by the size of each dimension or by the size (in bytes) of the element type (see Section 5.3.4).

Example:

TYPE

STRING ARRAY [26 BYTE]

INTERNAL

alpha STRING

!The array identifier "alpha" is defined as a 26-byte array!

is explained in detail in Section 5

LD R5, #0

LDB alpha(R5), #'A'

LDB alpha+l, #'B'

LDB alpha[2], #'C'

!The first element of array alpha now contains 'A', the second element contains 'B', and the third element contains 'C'!

Array definition and initialization are explained in detail in Section 5.3.4.

Record Structures. A record structure is a collection of named fields. Unlike array elements, record fields are not required to have the same type. For example, a record structure named "strobe" might have a BYTE field named "pin" and a SHORT_INTEGER field named "voltage".

TYPE

strobe RECORD [pin BYTE voltage SHORT_INTEGER]

sl strobe amount of the strong strong

As this example indicates, a RECORD type definition specifies an identifier and a type for each of its fields, as well as an identifier for the record structure itself. Individual fields can be referenced subsequently by specifying a record variable name followed by a period '.' and the field name.

LDB RL5, sl.voltage

The scope of a record identifier is specified in the variable declaration in which it is introduced (or it is implicitly INTERNAL if introduced in a TYPE statement). The scope of record field identifiers is the module or procedure in which they are introduced; in other words, all field identifiers must be unique in their entire scope. Record definition is explained in detail in Section 5.3.4.

<u>Label Type</u>. An identifier with type LABEL can only be used as a statement label. The LABEL type declaration is used primarily to explicitly specify a label's scope. The scope of a label is assumed to be the module in which it appears (that is, INTERNAL).

Therefore, statement labels which are to be accessible throughout the module need not be declared, although they can be explicitly declared INTERNAL for documentation purposes. If a label is to have GLOBAL, EXTERNAL, or LOCAL scope, however, it must be declared explicitly in a label declaration statement.

GLOBAL 28 benileb at "adola" sellinebi vasta sitt TRIG FUNCTION LABEL

Label declaration is explained in detail in Section 5.3.5.

5.3.3 Type Definition

The type-definition statement associates an identifier with a fixed type. These identifiers are assumed to have a scope of the current module (that is, INTERNAL scope).

Types are used primarily to categorize data and informally associate attributes or properties with the value of the data. Types ARRAY and RECORD provide an abbreviated template for structured data storage. Z8000 PLZ/ASM allows arbitrary association of types and data values with no type-compatibility restrictions.

The format of the type definition statement is:

TYPE

[type_identifier_type]*

where

type identifier

conforms to the rules for identifiers (Section 2.2.1).

type is BYTE, SHORT_INTEGER,
INTEGER, WORD, LONG, LONG_INTE
a previously-defined type iden INTEGER, WORD, LONG, LONG INTEGER, a previously-defined type identifier, or is an ARRAY or RECORD type definition.

Example:

TYPE

limitl BYTE limit2 LONG

LIST ARRAY [128 BYTE]
matrix ARRAY [10 10 SHORT_INTEGER]
patient RECORD [age height weight BYTE
room WORD
sex BYTE]

char BYTE
letter char
digit char

5.3.4 Variable Declaration

The variable declaration statement is used to declare the type of variable identifiers and optionally to define their initial values. Its basic format is:

variable_identifier+ type [':=' initial value]

This format is limited, however, by the scope and type specified.

Initialization. There are several rules and special symbols used for initializations in general. When a single declaration contains more than one variable identifier, the corresponding initialization values are listed within square brackets. A simple variable is initialized using a constant expression. A structured variable is initialized using a "constructor", which is simply a list of values enclosed by square brackets, with each level of nesting within the structure denoted by a matching set of brackets. A special symbol, '...', indicates that the immediately preceding value or constructor is to be repeated for the rest of the variables at the current level of nesting. The special symbol '?' can be used as a placeholder in a list of initial values for simple variables or components of simple type within a structure, and indicates that the corresponding simple variable remains unassigned. An empty constructor, '[]', indicates that the corresponding structured variable remains unassigned.

Scope. The scope of the variable declaration can be GLOBAL, EXTERNAL, INTERNAL, or LOCAL (Section 4.2.5). The first three can be specified only in variable declarations at the module level. LOCAL can be specified only at the procedure level.

An EXTERNAL variable declaration cannot include an initial value. The type of the variable can be any simple or structured type, or a previously-specified type identifier. Consequently, the format of the variable declaration in this case reduces to:

variable_identifier+ type

Example:

EXTERNAL
counter WORD
input, output ARRAY [72 BYTE]
customer name STRING

Simple Variable Declaration. Simple variables are variables whose type is BYTE, SHORT_INTEGER, WORD, INTEGER, LONG, LONG_INTEGER, or a previously specified simple type identifier (Section 5.3.2). Simple variables which are GLOBAL, INTERNAL, or LOCAL may optionally be given an initial value.

Simple variables are initialized in one of two ways: with a single constant expression, or with a list of constant expressions enclosed in square brackets. In the first case, only one variable may appear in the declaration and is initialized to the constant value. If a list is supplied, the variables are initialized in left-to-right order from the initial-value list. The initial-value list may have fewer items than the variable-identifier list, but an error results if the initial-value list is longer.

INTERNAL HUE BYTE limit WORD := %FFFF total, subtotal BYTE := [0...]
A, B, X BYTE := ['A', 'B', 'X']
D, E, F BYTE := [0, 1] !D=O, E=1, F is still undefined!

Array Variable Declaration. An array variable is one whose type is ARRAY. An array variable declaration has the format:

> array identifier+ ARRAY '[' dimension+ type ']' [':=' initial value]

or:

array identifier+ array type [':=' initial value]

array_identifier conforms to the rules for identifiers (Section 2.2.1).

specifies the number of dimensions in the array structure and the number of elements in each dimension.
The dimension(s) must be one
or more constant expression(s)
(Section 2.3.2) or a single elements in each dimension. asterisk (*), as detailed below.

type

is BYTE, SHORT INTEGER, INTEGER, WORD, LONG, LONG INTEGER, a previously defined type identifier, or is an ARRAY, or RECORD type definition.

array type

is a previously-defined ARRAY type identifier.

initial value is a bracketed list of constant expressions, or a character sequence as detailed below.

Several array identifiers may appear in a single declaration, and may optionally be initialized if they are declared GLOBAL, INTERNAL, or LOCAL.

Array structures are initialized in left-to-right order from the initial values supplied and in row-major sequence (that is, in the sequence of ascending memory addresses). For example, a 3x3 matrix would be ordered in the following sequence:

[0,0] [0,1] [0,2] [1,0] [1,1] [1,2] [2,0] [2,1] [2,2]

Array structures are initialized by a bracketed list of constant expressions. If the initial-value list contains "n" items, the first "n" elements of the array structure are initialized (in row-major sequence). The number of constants supplied cannot exceed the number of elements in the array structure.

Example:

INTERNAL Indianamib

matrix ARRAY [10 10 LONG INTEGER] list ARRAY [10 BYTE] := [0,1,0,0,1]
TABLE ARRAY [4 BYTE] := ['T', 'O', 'D', 'S'] ONEDIM1, ONEDIM2 ARRAY [2 BYTE] := [[1...][2...]]

Normally, each dimension specified in the array declaration must be a constant expression so that variable upper bounds are prohibited. The sole exception is an array declaration initializing a one-dimensional array structure. In this case, '*' is specified as the dimension and the length of the list is determined by the number of items in the initialization list. When the '*' feature is used, array structures can be initialized in two ways: with a bracketed list of constants as described above, or with a character sequence (enclosed in single quote marks). In the latter instance, the array elements must be type BYTE, SHORT INTEGER, or a user-defined 8-bit type and each byte is initialized to a single character value.

Example:

INTERNAL

list ARRAY [* BYTE] := [0,1,0,0,1]!This array is only 5 bytes; "list" array in last example is 10 bytes, although only 5 are initialized!

TABLE ARRAY [* BYTE] := 'TODS' !Compare to TABLE array in last example! Record Variable Declaration. A record variable is a variable whose type is RECORD. A record variable declaration specifies an identifier for the record as a whole and an identifier and type for each field within the record. A field type may be ARRAY or RECORD, as well as a simple type.

A record variable declaration has the format:

record identifier+ RECORD '[' [field identifier+ type]+ ']' [':=' initial value]

or:

record identifier+ record type [':=' initial value]

where

identifier conforms to the rules for identifiers

(Section 2.2.1).

is BYTE, SHORT INTEGER, WORD, INTEGER, LONG, LONG INTEGER, a previously-defined

type identifier, or is an ARRAY or

RECORD type definition.

is a previously-defined RECORD type record type

identifier.

initial value is a bracketed list of constant

expressions.

Several record identifiers may appear in a single declaration, and may optionally be initialized if declared GLOBAL, INTERNAL, Each record field is initialized in left-to-right or LOCAL. order from the values given in the initial list. A list of "n" values enclosed by square brackets may be given, with the first "n" fields being initialized. Having more constants than the total number of record fields is flagged as an error.

Example:

GLOBAL.

[age, height, weight BYTE person RECORD

birth RECORD [day, month, year BYTE]

salary WORD]

[length BYTE char ARRAY [50 BYTE]] :=[0[0]] MSG RECORD !length field and first byte of char array

are initialized to zero!

If an array or record appears within another array or record, then this nesting is represented by enclosing each level of initialization values within square brackets. Notice that in this case, the last record fields or array elements at each level

do not have to be specified. Furthermore, if more than one structured variable identifier appears in a single declaration, then the part of the initial value list corresponding to each structured variable must also be enclosed by square brackets.

TYPE

PATIENT RECORD [ROOM WORD

BIRTH RECORD [DAY, MO, YR BYTE]

SEX BYTE]

INTERNAL

FEMALE ARRAY [100 PATIENT] :=[[?,[],'F']...]
!only the SEX field of each record is initialized!

Alignment. If the current location counter value is even, then any data variable or instruction may start at that address. However, if the current location counter value is odd, then only 8-bit values may start at that address. The assembler automatically aligns simple variables so that word (WORD or INTEGER) or long word (LONG or LONG_INTEGER) variables are located at even addresses. This is accomplished by inserting a byte with an undefined value when the location counter value is odd, thus forcing the word or long word to start at the next higher even address. Instructions within a procedure are also automatically aligned.

Structured variables are, in general, automatically aligned; that is, all array or record variables are forced to start at an even address (an undefined byte value is inserted if the location counter was odd). An exception to this rule is made for arrays whose elements are any 8-bit type (based on BYTE or SHORT_INTEGER), which do not have to be aligned on an even address boundary.

It is the programmer's responsibility to ensure that all variables within a structure are aligned properly. The programmer should be careful to align record fields on their appropriate address boundaries relative to the start of the record. This can be accomplished by defining "filler" fields where necessary to force the alignment of the following field.

5.3.5 Label Declaration

The label declaration statement specifies that an identifier is used in the program as a statement label. It cannot be used for any other purpose within its defined scope. The format of the label declaration is:

label_declaration+ LABEL STYS FIGURE

where

label_identifier conforms to the rules for identifiers in Section 2.2.1.

Note that the colon (:) that follows a label identifier when it appears in an executable statement (Section 2.2.1) is not included when the label is identified in a label declaration statement. Note also that labels cannot be given an initial value (that is, the label declaration cannot be used to assign absolute addresses).

A label can have GLOBAL, EXTERNAL, INTERNAL, or LOCAL scope (Section 4.2.5). If a label is used in an executable statement without being declared in a label declaration statement, it is assumed to be INTERNAL to the module in which it appears. This default scope can be overridden by explicitly declaring the scope of the label.

Notice that a label with LOCAL scope must be declared in the procedure declaration before the ENTRY keyword; that is, before the label is used or defined, whereas a label with GLOBAL, EXTERNAL, or INTERNAL scope may be used before it is either declared or defined. To allow the programmer to avoid pre-declaring LOCAL labels, Z8000 PLZ/ASM provides a special form of statement labels which are always of LOCAL scope and cannot be declared explicitly. The form of a special label is a dollar sign "\$", followed immediately by any valid decimal number, and it can be used in the same manner as a regular statement label identifier, except that its scope is always limited to the procedure in which it is defined.

Example:

process10 PROCEDURE !Procedure has GLOBAL scope! LOCAL a,b,c BYTE declaration+ LABEL ledel STEP2 LABEL ENTRY !STEP1 has INTERNAL scope! STEP1: ... conforms to the rules for identifiers in Section 2.2 STEP2: ... STEP2: STEP2 is LOCAL to grant additional state of the stat

"process10"!

STEP3: ... !STEP3 has GLOBAL scope!

\$1: ... !\$1 is LOCAL to "process10"! without being declared in a label declaration statement, it is

END process10

5.3.6 SIZEOF Operator WHOME and another mothers and another mothers and another mothers and another mothers are not as a second mothers and another mothers are not as a second mother mother and a second mother mothers are not as a second mother mother mothers and a second mother mothers are not as a second mother mother mothers and a second mother mothers are not as a second mother mothers and a second mother mothers are not as a second mother mothers and a second mother mothers are not as a second mother mothers and a second mother mothers are not as a second mother mother mothers are not as a second mother mothers are not as a second mother mother mothers.

Z8000 PLZ/ASM includes a special unary operator, SIZEOF, which operates on type identifiers to determine the size (in bytes) of a variable field. Although SIZEOF can be used with any type, it is useful for sizing arrays, records, and user-defined types.

to be INTERNAL to the module in which it appears ...

declared explicitly. The form of a special label is a doller sign "5", followed immediately by any valid decimal SqYTer, and char BYTE digit char matrix ARRAY [10 10 WORD] patient RECORD [height weight BYTE room WORD]

> LD RO, #SIZEOF digit Rl, #SIZEOF matrix LD Rl, #(SIZEOF matrix/ SIZEOF WORD) R2, #SIZEOF patient

!RO contains 1! !Rl contains 200! !Rl contains 100! !R2 contains 4! LD R3, #SIZEOF patient.weight !R3 contains 1!

Appendix A High-Level Statement Summary

The following list summarizes the high-level keywords and their uses.

ARRAY One of the two structured variable types. Used in type definition and variable declaration statements.

One of the simple variable types.

Used in type definition and variable

declaration statements.

CASE

Used in IF conditional-execution statement. Instruction(s) following CASE definition are executed if one of the specified values matches

selector register.

CONSTANT Introduces constant definition(s).

DO Introduces DO loop.

ELSE Used in IF conditional-execution statement. Statements between ELSE

and FI are executed if the specified

condition is false.

END Module or procedure terminator.

ENTRY Marks beginning of action-statement

part of a procedure.

EXIT Loop control statement used to

control execution flow of a DO loop.

EXTERNAL Specifies that variables and/or

procedures defined as GLOBAL in another module will be used in the

current module.

FI IF statement terminator.

FROM Used in conjunction with EXIT and

REPEAT loop control statements.

GLOBAL Declares variables and/or procedures

to have a scope of the entire

executable program.

IF Introduces IF statement. Code

following IF-THEN is executed if the

specified condition is true.

INTEGER One of the simple variable types.

Equivalent to WORD.

INTERNAL Declares variables and/or procedures

to have a scope of the current module

only.

LABEL Used to declare statement label

scope explicitly.

LOCAL Declares variables to have a

scope of the current procedure only.

LONG One of the simple variable types.

Used in type definition and variable

declaration statements.

LONG INTEGER One of the simple variable types.

Equivalent to LONG.

MODULE Introduces a module.

OD DO loop terminator.

PROCEDURE Introduces a procedure.

RECORD One of the two structured variable

types. Used in type definition and

variable declaration statements.

REPEAT Loop control statement used to

control execution flow of a DO loop.

SHORT_INTEGER One of the simple variable

types. Equivalent to BYTE.

THEN Used in IF conditional-execution

block. The statements between THEN

and ELSE (or THEN and FI if ELSE is

omitted) are executed if the specified condition is true.

TYPE

Introduces type definition(s).

WORD

One of the simple variable types. Used in type definition and variable declaration statements.

The remainder of this appendix contains the complete grammar for Z8000 PLZ/ASM. In this grammar:

- Keywords are shown as all uppercase letters;
- Required special characters are enclosed in 'single quotes';
- Optional items are enclosed in [square brackets];
- Possible repetition of an item is shown by appending a "+" (to signify one or more repetitions) or "*" (to signify zero or more repetitions) to the item;
- Parentheses are used to group items to be repeated;
- A vertical bar "|" signifies that an alternative follows.

PLZ/ASM GRAMMAR - MODULE SYNTAX

module

=> module_identifier MODULE declarations* END module identifier The remainder of this appendix contains the complete grammar for

declarations

=> constants => types => globals => internals => externals

constants => CONSTANT constant definition*

types => TYPE type_definition*

globals => GLOBAL var or proc declaration*

internals

=> INTERNAL var_or_proc_declaration*

externals

=> EXTERNAL restricted_var_or proc_declaration*

constant definition

=> constant_identifier ':=' expression

expression

=> arithmetic expression [rel op arithmetic expression] *

arithmetic expression

=> term [add op term] *

term

=> factor [mult op factor]*

factor

=> unary_operator factor => '(' expression ')' => SIZEOF type_identifier => constant identifier

=> label => variable => number

=> character constant

character constant => character sequence type definition => type identifier type type => simple type => structured type simple type => BYTE => WORD => LONG => SHORT INTEGER => INTEGER => LONG INTEGER => simple type identifier structured type => array type => record type => ARRAY '[' expression+ type ']' array type => array type identifier record type => RECORD '[' field declaration+ ']' => record type identifier field declaration => field identifier+ type var or proc declaration => variable declaration => procedure declaration variable declaration => variable noinitial declaration => variable initial declaration => label declaration restricted var or proc declaration => identifier+ type => procedure identifier PROCEDURE variable noinitial declaration => identifier+ type

```
variable initial declaration
                       => identifier simple type
                          ':=' initial value
                       => identifier identifier+ simple type
                          ':=' '[' initial_value* ['...'] ']'
                       => identifier structured type
                          ':=' constructor
                       => identifier identifier+ structured type
                        ':=' '[' constructor*
                       => identifier ARRAY
                          '[' '*' simple type ']'
                          ':=' '[' initial value+ ']'
                       => identifier ARRAY
                          '[' '*' simple_type ']'
                          ':=' character sequence+
                       => '[' initial_component*
constructor
                         ['...'] ']'
initial component => initial_value
                     => constructor
initial value
                     => expression
                       => 1?1
                      => label identifier+ LABEL
label declaration
variable
                       => identifier
                       => array_variable
                       => record variable
array variable
                       => array designator
                          '[' expression+ ']'
                       => array identifier
array designator
                       => record variable
                       record variable
                       => record designator
                          '.' field identifier
                       => record identifier
record designator
                       => array_variable
                       => record variable
```

```
procedure declaration => procedure identifier
                          PROCEDURE
                             locals*
                          [ENTRY
                            statement*1
                        END procedure identifier
locals
                   => LOCAL
                           variable declaration*
statement
                       => [label ':'] statement
                       => loop statement
                      => exit statement
                      => repeat statement
                      => if statement
                     => select_statement
                       => assembler instruction
loop statement
                      => DO
                               statement*
                         OD
label
                      => label identifier
                      => '$' decimal constant
exit statement
                      => EXIT [FROM label]
repeat statement
                      => REPEAT [FROM label]
                   => IF condition code THEN statement*
if statement
                              [ELSE statement*] FI
                      => IF selector_register
select statement
                              select element+
                              [ELSE statement*] FI
selector register
                      => register
select element
                      => CASE expression+
                              THEN statement*
                      => '= ' | '<> ' | '< ' | '> ' | '<= ' | '>= '
rel op
                      => '+' | '-' | LOR | LXOR
add op
                      => '*' | '/' | LAND | MOD | SHL | SHR
mult op
                => '+' | '-' | LNOT
unary operator
assembler instruction => operation operand*
operation
                     => Z8000 instruction
```

```
operand
                   => register
                        => indirect register
                        => immediate
                        => indexed address
                       => based address
                        => based_indexed_address
                        => direct address
                        => relative address
                       => condition code
                        => flags
                        => int
register
                      => single register
                      => double register
                        => low byte register
                     => high byte register
                        => quad register
                        => special register
                        => R0 | R1 | R2 | R3 | R4 | R5 |
single register
                            R6 | R7 | R8 | R9 | R10 | R11 |
                            R12 | R13 | R14 | R15
double register
                        => RR0 | RR2 | RR4 | RR6 |
                            RR8 | RR10 | RR12 | RR14
low byte register
                         => RL0 | RL1 | RL2 | RL3 |
                         RL4 | RL5 | RL6 | RL7
high byte register
                        => RH0 | RH1 | RH2 | RH3 |
                            RH4 | RH5 | RH6 | RH7
quad register
                         => RQ0 | RQ4 | RQ8 | RQ12
indirect register
                         => '@' register
                        => '#' expression
=> '#' SEG expression
immediate
                         -> '#' OFFSET expression
indexed address
                         => address designator
                            '(' register ')'
based address
                        => register '(' immediate ')'
based indexed address => register '(' register ')'
direct address
                        => address designator
                        => address designator
relative address
```

address_designator => '|' seg_address_designator '|' => seg_address_designator

seg_address_designator => expression

=> '<<' expression '>>' expression

flags | Man | Man | State | Karkasa

SDLL | SET | SETE | SETFLC | SIN |

SINB | SIND | SINDB | SINDR | SINDR |

SINI | SINIB | SINIR | SINIRB | SEA |

SINB | SLAL | SLL | SLL | SLL |

SOTDR | SOTDR | SOTIR | SOUTE |

SOUT | SOUTE | SOUTE | SOUTE |

SOUT | SOUTE | TOUT | SOUTE |

TESTE | TESTE | TESTE |

TRIER | TRTDB | TRTDB | TRTIB |

TRTIRB | TSET | TSETE | XOR | XORB

INT | IVN | IV <= REE | MEET | MULT | INI | INI | INI | IVN | IV <= REE | MEET | MULT | MULT

Z8000 instruction => ADC | ADCB | ADD | ADDB | ADDL | AND | ANDB | BIT | BITB | CALL | CALR | CLR | CLRB | COM | COMB | COMFLG | CP | CPB | CPL | CPD | CPDB | CPDR | CPDRB | CPI | CPIB | CPIR | CPIRB | CPSD | CPSDB | CPSDR | CPSDRB | CPSI | CPSIB | CPSIR | CPSIRB | DAB | DBJNZ | DEC | DECB | DI | DIV | DIVL | DJNZ | EI | EX | EXB | EXTS | EXTSB | EXTSL | HALT | IN | INB | INC | INCB | IND | INDB | INDR | INDRB | INI | INIB | INIR | INIRB | IRET | JP | JR | LD | LDA |
LDAR | LDB | LDCTL | LDCTLB | LDD | LDDB | LDDR | LDDRB | LDI | LDIB | LDIR | LDIRB | LDK | LDL | LDM | LDPS | LDR | LDRB | LDRL | MBIT | MREO | MRES | MSET | MULT | MULTL | NEG | NEGB | NOP | OR | ORB | OTDR | OTDRB | OTIR | OTIRB | OUT | OUTB | OUTD | OUTDB | OUTI | OUTIB | POP | POPL | PUSH | PUSHL | RES | RESB | RESFLG | RET | RL | RLB | RLC | RLCB | RLDB | RR | RRB | RRC | RRCB | RRDB | SBC | SBCB | SC | SDA | SDAB | SDAL | SDL | SDLB | SDLL | SET | SETB | SETFLG | SIN | SINB | SIND | SINDB | SINDR | SINDRB | SINI | SINIB | SINIR | SINIRB | SLA | SLAB | SLAL | SLL | SLLB | SLLL | SOTDR | SOTDRB | SOTIR | SOTIRB | SOUT | SOUTB | SOUTD | SOUTDB | SOUTI | SOUTIB | SRA | SRAB | SRAL | SRL | SRLB | SRLL | SUB | SUBB | SUBL | SWAP | TCC | TCCB | TEST | TESTB | TESTL | TRDB | TRDRB | TRIB | TRIRB | TRTDB | TRTDRB | TRTIB | TRTIRB | TSET | TSETB | XOR | XORB

PLZ/ASM GRAMMAR - LEXICAL SYNTAX

```
PLZ text
id constant text
                        => identifier
                        => constant
                        => delimiter text
separator
                        => special symbol
identifier
                        => letter (letter | digit | '')*
constant
                        => number
                        => character sequence
number
                        => decimal constant
                        => hex constant
                        => octal constant
                        => binary constant
decimal constant
                       => digit+
hex constant
                       => '%' hex digit+
octal constant
                       => '%(8)' oct digit+
binary constant
                       => '%(2)' bin digit+
character sequence
                        => ''' string text+ '''
string text
                        => string char
                        => special string text
string char
                        => any character_except % or '
special string text
                        => '%' special string char
                        => '%' hex digit hex digit
                        => 'R' | 'L' | 'T' | 'P' | 'Q' | '%' | 'r' | 'l' | 't' | 'p' | 'q'
special string char
                        => 'A' | 'B' | ... | 'Z'
letter
                           'a' | 'b' | ... | 'z'
                        => '0' | '1'
bin digit
oct digit
                        => '0' | '1' | '2' | '3' |
                           141 | 151 | 161 | 171
```

```
digit => '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'
                      => '0' | '1' | '2' | '3' |
hex digit
                         '5' | '6' | '7' | '8' | '9' |
'A' | 'B' | 'C' | 'D' | 'E' | 'F' |
    'a' | 'b' | 'c' | 'd' | 'e' | 'f'
                      => '+' | '-' | '*' | '/' | '...' |
special symbol
                     => delimiter
delimiter text
                      => comment
                      => '!' any character except ! '!'
comment
                   => ';' | space | ',' |
delimiter
                     tab | formfeed | linefeed |
                  carriage return
```

Appendix B Assembler Directives and **Extended Instructions**

B.1 ASSEMBLER DIRECTIVES

The following is a summary of some of the assembler directives used to control the operation of the Z8000 assembler. Other directives are explained in detail in the Z8000 Assembler User's Guide.

These directives can be embedded in the source program, and always start with a dollar sign "\$", immediately followed by the particular directive and then any operands. For example, a programmer may want to fix the address for a procedure:

\$ABS 12

The assembler directives are:

\$ABS [location]

Specifies that the object code produced is to be absolute. If "location" is specified, the location counter will be set to that value and the assembler and another ward 00088 will begin assigning absolute addresses from that location. "Location" "ist be a constant expression. \$ABS rema is in effect until a \$REL directive is encountered.

\$REL [location] Specifies that the object code produced is to be relocatable. If "location" is specified, it is a relocatable offset from the beginning of the current section. "Location" must be a constant expression. \$REL remains in effect until SABS directive is encountered. \$REL 0 is the default at the start of module assembly.

SSDEFAULT

Cancels the effect of the \$SECTION assembler directive and restores default memory assignment. Data declarations reside in the data section and procedure declarations reside in the program section.

\$SECTION identifier Causes the object code produced to be associated with a symbolic identifier which can be used later for mapping into one of the memory areas. Remains in effect until either another SSECTION directive is encountered, or until a \$SDEFAULT directive restores default memory assignment.

B.2 EXTENDED INSTRUCTIONS

The following three "extended" instructions may be placed within a procedure as a one-operand assembly language instruction and produce an arbitrary byte, word, or long word value.

BVAL constant expression

Defines a byte value to be located at the current location counter. Can be used to "create" Z8000 instructions or

WVAL constant expression Defines a word value to be May as to be a selected at the current location counter. Can be used to counter 28000 instructions or data.

LVAL constant expression Defines a long word value to be located at the current location counter. Can be used to because about about and sent "create" Z8000 instructions or nolyspol at aldatspoler data.

Appendix C Reserved Words and Special Characters

C.1 RESERVED WORDS

Certain special symbols are reserved for Z8000 PLZ/ASM and can not be redefined as symbols by the programmer. These are the names of operators, condition codes, register symbols, assembly language instructions, and high-level statement keywords. The specific reserved words are listed below.

NAMES OF OPERATORS

| LAND | OFFSET |
|------|--------|
| LNOT | SEG |
| LOR | SHL |
| LXOR | SHR |
| MOD | SIZEOF |
| | |

STATUS FLAGS AND CONTROL BITS

| C | V |
|-----|----|
| NVI | VI |
| P | Z |
| S | |

CONDITION CODES

| C | LE | NE | PE | UGT |
|----|----|-----|-----|-----|
| EQ | LT | NOV | PL | ULE |
| GE | MI | NZ | PO | ULT |
| GT | NC | OV | UGE | Z |

CONTROL REGISTER SYMBOLS

| FCW | PSAP |
|--------|---------|
| FLAGS | PSAPOFF |
| NSP | PSAPSEG |
| NSPOFF | REFRESH |
| NSPSEG | |

ASSEMBLY LANGUAGE INSTRUCTIONS

| ADC ADCB ADCB ADDD ADDB ADDL AND ANDB BIT BITB CALL CALR CLR CLRB COM COMB COMFLG CP CPB CPL CPD CPDB CPL CPDR CPDR CPDR CPDR CPIR | CPSD CPSDB CRSDR CPSDRB CPSIB CPSIR CPSIRB DAB DBJNZ DEC DECB DI DIV DIVL DJNZ EI EX EXB EXTS EXTS EXTSB EXTSL HALT IN INB INC | IND INDB INDR INDR INIB INIR INIR INIRB IRET JP JR LD LDA LDAR LDB LDCTL LDCTLB | LDM LDPS LDR LDRB LDRL MBIT MREQ MRES MSET MULT MULTL NEG NEGB NOP OR OTDR OTDR OTDR OTIR OUTIR OUTD OUTDB OUTI OUTIB | POPL PUSH PUSHL RES RESS RESFLG RET RL RLB RLC RLCB RLCB RLDB RR RRB RRC RRCB RRCB RRCB RRCB RRCB R | SETB SETFLG SIN SINB SIND SINDR SINDR SINDR SINIR SOTOR | SOUTIB SRA SRAB SRAL SRLL SRLB SRLL SUB SUBB SUBL TCC TCCB TEST TESTB TESTL TRDB TRDRB TRIB TRIRB TRIDB TRITB TRITB TRTDB TRTDB TRTDB TRTDB TRTDB TRTDB TRTDB TRTDB |
|--|--|--|---|---|---|---|
| | | | | | | |

When defining symbols, users should also avoid the forms Rn, RHn, RLn, RRn, and RQn where n is a number from 0 to 15.

EXTENDED INSTRUCTIONS

BVAL LVAL WVAL

HIGH-LEVEL STATEMENT KEYWORDS

| ARRAY | END | GLOBAL | LONG | REPEAT |
|----------|----------|----------|--------------|---------------|
| BYTE | ENTRY | IF | LONG INTEGER | SHORT INTEGER |
| CASE | EXIT | INTEGER | MODULE | THEN |
| CONSTANT | EXTERNAL | INTERNAL | OD | TYPE |
| DO | FI | LABEL | PROCEDURE | WORD |
| ELSE | FROM | LOCAL | RECORD | |

C.2 SPECIAL CHARACTERS

The list of special characters below includes delimiters and special symbols. The difference between them is that delimiters have no semantic significance (for example, two PLZ/ASM tokens can have any number of blanks separating them), whereas special symbols do have semantic meaning (for example, # is used to indicate an immediate value).

The class of <u>delimiters</u> includes the space (blank), tab, form feed, line feed, carriage return, semicolon (;), and comma (,). The comment construct enclosed in exclamation points (!) is also considered a delimiter.

The special symbols and their uses are as follows:

field reference.

| + | Binary addition; unary plus. |
|----|---|
| - | Binary subtraction; unary minus. |
| rk | Unsigned multiplication; dimension specifier for list (one-dimensional array) initialization. |
| / | Unsigned division. |
| • | Label terminator. |
| := | Constant and variable initialization. |
| % | Nondecimal number base specifier; special character specifier within quoted character sequence. |
| # | Immediate data specifier. |
| @ | Indirect address specifier. |
| \$ | Current contents of location counter; specifies special LOCAL statement labels; precedes assembler directives. |
| [] | Enclose components of ARRAY or RECORD definition; enclose index part of ARRAY reference; enclose initialization values. |
| () | Enclose expressions selectively; enclose octal or |

binary number base indicator; enclose index part of indexed, based, and based indexed address.

Separates RECORD name from field name in RECORD

1 "Less than" operator "Greater than" operator. > "Equal" operator. BARAL MEAL XII **<>** "Not equal" operator. <= "Less than or equal" operator. "Greater than or equal" operator. >= Placeholder in initialization lists. Repetition symbol in initialization lists. ((>> Denotes segmented address. 1 1 Enclose short offset segmented address.

urrent contents of location counter; apecifies pecial LOCAL statement labels; precedes essembler directives.

Inclose components of ARRAY or RECORD definition inclose index part of ARRAY reference; enclose nitialization values.

Inclose expressions selectively; enclose octal context number base indicator; enclose index part indexed address.

Introduction to RIO Text Processing

CONTENTS

| SECTION | 1 | INTRODUCTION 30' | | |
|---------|---|--|--|--|
| SECTION | 2 | GETTING STARTED 308 | | |
| | | 2.1 The Computer System | | |
| SECTION | 3 | USING THE RIO TEXT EDITOR 315 | | |
| | | 3.1 Introduction | | |
| SECTION | 4 | DISK AND FILE KNOW-HOW 323 | | |
| | | 4.1 Formatting a New Disk 323 4.2 Copying a Disk 324 4.3 Copying a File 324 4.4 Listing the Directory of Files 325 4.5 Deleting a File 325 | | |
| SECTION | 5 | GOOD THINGS TO KNOW 326 | | |
| | | 5.1 RIO Editor Command Summary | | |

SECTION 1

INTRODUCTION

Using a computer text editor to prepare a document is similar to using a typewriter. The difference is that with the computer, making changes to text is much easier.

The purpose of this document is to give a general overview of how to use the RIO text editor. RIO is an interactive computer system, meaning you and the system can communicate with each other by means of a keyboard terminal electronically connected to the computer.

This document has been designed with the beginner in mind. Those already familiar with text editing may wish to start with the Z-80 RIO Text Editor User's Manual (see Section 5). After you have learned the fundamentals of RIO text editing, refer to other Zilog documents (see Section 5), which have been designed for the more experienced user.

Learning to use the RIO text editor is like learning anything else -- once you become familiar with it, it will seem easy. Try to spend a few uninterrupted hours getting to know your system. As you read through this manual, put into practice what you learn. Don't just read about it, try it.

SECTION 2

GETTING STARTED

2.1 The Computer System and the days and designed as part

The first step in learning about RIO text editing is to become familiar with your computer system.

There are three basic parts:

1. The computer: The basic unit of your computer system
 (see Figure 1) is where all the information is processed
 -- it is the "brain" of your system.

The information necessary for the computer's operation is stored on magnetic disks. These are housed in protective, square coverings with window cut-outs that expose the disk. The disks are inserted into two "disk drives" to receive information from, or supply information to, the computer. The disk that contains information necessary to run RIO is called the System Disk, and usually goes into the rightmost drive (Drive O). The disk on which your data is stored is called the User Disk. It is inserted into the left drive, which is Drive 1 or 2, depending on your system.

The type of system you have appears on the front panel of your computer. In a Microcomputer Zilog (MCZ), the left drive is called Drive 2. In a Zilog Development System (ZDS), the left drive is called Drive 1. In this manual the left drive will always be referred to as Drive 2. If you have a ZDS, don't forget to always use "1" when "2" is specified.

CAUTION

When handling disks, be sure not to touch the exposed window areas. This could damage information stored on the disk.

2. The keyboard: A keyboard similar to a typewriter is used to enter information into the computer. Note that the numerical keys "0" and "1" (zero and one) can never be interchanged with the uppercase letter "0" and lower case letter "1", respectively when giving commands to the system.

The key marked RETURN is the carriage return. After a command to the computer is given, this key must be pressed to execute the command.

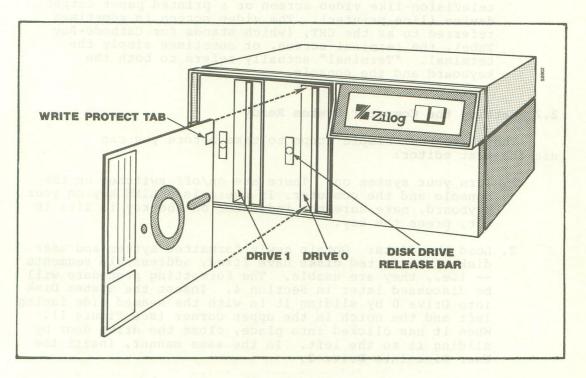


Figure 1. Zilog Computer System

The CAPS ONLY key is similar to the shift lock on a typewriter keyboard. The difference is that when the CAPS ONLY light is on, only letters are capitalized. To type special symbols (!, ", %, \$, etc.) you must still press SHIFT while pressing the appropriate key.

To correct a typing mistake press the BACKSPACE key. This deletes the last character typed. On keyboards without the BACKSPACE key, the last character is deleted by holding down the CNTL (control) key while striking the "H" key.

Pressing the DEL (or RUBOUT) key, erases the entire current line. The current line is the last line typed.

3. The console: As you type on the keyboard, what you type is displayed on a console. This is either a television-like video screen or a printed paper output device (line printer). The video screen is sometimes referred to as the CRT, (which stands for Cathode-Ray Tube), the terminal screen, or sometimes simply the terminal. "Terminal" actually refers to both the keyboard and the console.

2.2 Getting the Computer System Ready

There are four simple steps to take before you can use the text editor:

- Turn your system on: There are on/off switches on the console and the computer. If there is a LINE key on your keyboard, make sure the indicator on the key is lit; if not, press the key.
- 2. Load the disks: Obtain newly formatted system and user disks. Formatted disks have fixed, addressable segments i.e., they are usable. The formatting procedure will be discussed later in Section 4. Insert the System Disk into Drive 0 by sliding it in with the seamed side facing left and the notch in the upper corner (see Figure 1). When it has clicked into place, close the drive door by sliding it to the left. In the same manner, insert the User Disk into Drive 2.
- 3. Bootstrap: Bootstrapping gets the computer ready to accept commands. Do this by pressing the RESET button on the computer and the RETURN on the keyboard. A successful Bootstrap results in a "%" on the terminal screen. This is called a "prompt", and means the computer is ready to accept commands.
- 4. Initialize: Type an "I" after the prompt and strike the carriage return.
- NOTE: The Initialize command should always be the first command entered after you insert a disk. This step will not be indicated throughout the manual, so it is important to get into the habit of typing an "I", followed by a carriage return, each time you insert a disk.

2.3 Getting Information into the Computer (Files)

Information is stored on the disks in files. Imagine files as individual documents, which you call forth at will to change or print into a hard-copy (computer slang for the actual ink-on-paper representation of what a file contains). Each file has its own name, so it can be easily located by the computer.

Creating a new file on the computer is like creating a new document on a typewriter. The disk holds the information you enter on the keyboard just as a piece of paper in a typewriter holds what you type.

You can easily obtain a hard-copy of a file with a line printer. Most systems are not connected to a printer, but there is usually one somewhere close by. There are many kinds of printers, so when you are ready to use one you will need to consult your local electronics wizard to learn how.

2.4 Creating a File

You are now ready to create a file. The first one you create, by following the steps in the manual, will be simple and short, but it will give you a good idea of how to create your own files in the future. The name of this file will be APHORISMS.

Make sure the prompt (%) sign is visible on your screen; if not, Bootstrap.

There will be two things going on in all of the following instructions -- you will be typing on the keyboard to the computer, and the computer will be answering you on your console.

In all of these instructions, what you are supposed to type will be in **boldface**, what the computer responds is not boldfaced, and a right arrow ("--->") will represent a carriage return.

Most commands to the computer are typed in uppercase letters.

With all of this in mind, type:

% EDIT APHORISMS --->
EDIT 1.6
NEW FILE
INPUT

When you type EDIT APHORISIMS, you are specifying an action and naming a file. The computer searches its memory (searches through all the pre-existing files), and, in this case, because it cannot locate a file named APHORISMS, it states that APHORISMS is a new file. Don't be concerned if your computer responds with some number other than 1.6 after EDIT, as this is simply a release number for your System Disk.

When you create a file, you use the RIO Text Editor. "Edit" is a function on your RIO system that makes it easy to prepare written material for presentation or publication. To edit a file, you must be in "Edit mode".

When you type "EDIT APHORISMS" you are put in Edit mode. When the file you are editing is a new file, as in this case, the computer goes one step further and puts you in "Input mode". Input means that you may now add text to a file. The computer tells you that you are in Input mode by not issuing a prompt. If the last line on your console contains a ">", type "INPUT" and a carriage return.

CAUTION

The Percent-Sign prompt (%) has disappeared from the last line, so do not remove your disks. Only remove them when the "%" prompt is present and the red lights on the disk release bars are off. Removing disks at the wrong time can cause file damage.

When entering the aphorisms that follow, insert carriage returns so your aphorisms look the same as those in this manual. Carriage returns will not be indicated here for the straight input of text.

Type into the computer:

A penny saved is a penny earned.

You will often want to skip spaces between paragraphs or sections when inputting text. You can't, however, accomplish this by repeatedly pressing the carriage return. Instead, for each additional blank line you want, press the carriage return once, touch the long space bar (character space) at the bottom of the keyboard, and then press the carriage return again. If you type two successive carriage returns you will go out of Input mode. If a ">" appears before you are done inputting, type "I", carriage return, and continue where you left off. Typing "I" after a right angle bracket prompt (>) means "put me in Input mode". Note that this is different from typing an "I" after a percent sign prompt (%), which means "Initialize".

Double-space after your last aphorism, and add these golden words to your collection:

A stitch in time saves nine.

Into each life a little rain must fall.

Every cloud has a silver lining.

A rolling stone words we sale to the day of the sale to the sale t

You can lead a horse to water, the same same but you can't make it drink.

2.5 Closing a File sanster wath and no adopt her add light the

To close the aphorisms file, you must go from Input mode back into Edit mode. You entered Edit mode when you opened the file. The ">" prompt indicates that you are in Edit. Close your file by hitting a carriage return twice. On your console will appear:

EDIT

To get out of Edit mode, type: daddgmoo and flat all avenue and

*X * S=0 D=2 APRORISMS ---> <-- TIUQ<

When the prompt is present and the red lights on the disk release bars are off, remove the disks by pressing the bars — the disks will pop out.

This is a good time to make a hard-copy of your APHORISMS file if you have a printer available.

2.6 Creating a Backup for Files 11 9038 Baddall boa edd wells

You will shortly make changes to your APHORISMS file, but before doing so, you should make a backup disk. Keep this disk only for backing up files. In this way you will have an exact copy of your files on hand in case anything happens to the originals. It is a good idea to back-up frequently. At some point (like when you've worked for days on a deadline project and then accidently erased your entire file), you will be very glad you did.

Making a back-up is simple. First obtain a user disk and label it "BACK-UP". Make sure there is a silver "Write Protect" tab on the notch in the disk (see Figure 1).

If you removed your System Disk to make a hard copy, put it back into Drive 0 and Bootstrap. The "%" prompt should be present. Put your back-up disk into Drive 2. If you can give the Initialize command without the computer responding with an "Error C4" message, your user disk is formatted, therefore it is usable. If you can't initialize without receiving an error message, type:

%FORMAT D=2 --->
DISK ID: BACKUP
READY? Y

Wait until the red lights on the disk release bars are out and the "%" prompt is present before continuing.

You are now ready to move a copy of the Aphorisms file to the back-up disk. Type the following command, making sure not to leave out the comma:

% MOVE, --->

Remove the System Disk and put the disk with your Aphorisms file into Drive 0. Tell the computer to copy your file by typing:

%X * S=0 D=2 APHORISMS ---> X * S=0 D=2 APHORISMS

FILENAME DRIVE
APHORISMS 0

S stands for source, and D stands for destination. If your system gave a "Destination Device Not Ready" message, you probably have a ZDS system and didn't change "D=2 to "D=1".

When the red lights stop flashing and the prompt sign appears, the file has been copied. Put the System Disk back into Drive 0 and the disk with the Aphorisms file back into Drive 2. Don't forget to Initialize.

There is also a way to copy an entire disk if you wish. This is covered later in Section 4.

SECTION 3

USING THE RIO TEXT EDITOR

3.1 Introduction

In this section you will modify your APHORISMS file and see first-hand how the RIO Editor works. If you make a mistake, finish reading the subsection you're in. This should help clear up what to do to fix the mistake.

3.2 Getting into Edit 1100x9 vino 918001 liw 1915 may 1

To get back into your APHORISMS file in the Edit mode, type:

*EDIT 1.6

Compare this with the interaction with the computer when a new file was created. Since APHORISIMS already exists, the system does not inform you that it is a new file, nor does it automatically put you in Input mode. Note that you must be editing a file before you can get into Input mode. If a "%" prompt is present, you cannot get into Input without first specifying the file you wish to input text to.

3.3 Locating a String: The "L" Command

If you wish to locate a specific part of your text, you can use the Locate command by typing an "L", followed by the string you wish to find. A string is a sequence of characters set aside by some separator. This document usually uses slash marks as string separators.

As an example, locate "Every cloud has a silver lining."

Type:

>L /cloud/ --->
Every cloud has a silver lining.

The computer searches the Aphorisms file from its current location downward, until it comes to the first occurrence of what is contained within the slash marks. Since you just started this editing session, the current location of the Editor (before you gave the Locate command) was the top of the file, or the first line of the first page.

When giving the Locate command, use strings that are unique (in this case, "cloud"); otherwise, your search will be inefficient, as RIO stops at the first occurrence of the desired string. If there are many occurrences of the given string, you might have to repeat the command several times before you find the desired line.

The computer will locate only exactly what you ask it to locate. If the computer responds with "EOF", it is at the end of the file, and hasn't found your string. If so, type "T" and carriage return, then try again. Make sure you haven't added an extra space, misspelled something, or capitalized when you shouldn't have.

3.4 Making a Change: The "C" Command

The Change command instructs RIO to change what is contained within one set of string separators to that which is contained within the next set -- C /change this/to this/. Make sure to leave a space after the "C".

The following line should be on your console:

Every cloud has a silver lining.

Change this line by typing:

>C /cloud/Rolls Royce/ --->
Every Rolls Royce has a silver lining.

Note that you can use the Change command to delete a string. Do this by not specifying anything between the last set of separators. For example, if you want to delete the word "Royce", you could type "C / Royce//".

3.5 Locating a String: The "P" and "U" Command

There is more than one way to locate a string. This time you will locate "gathers no moss", but by using the Print and Up commands.

The Print command writes a specified number of lines on the console, starting from the current line. Because you are approximately five lines from "gathers no moss", type:

>P 5 ---> Every Rolls Royce has a silver lining.

A rolling stone many seems seems and the seems of the see

>

Note that your current line, (the line above the ">" prompt), is a blank line, and the line you are looking for is the line above your current line. Use the Up command to get there by typing:

>U 1 ---> gathers no moss.

3.6 Replacing a Line: The "R" Command

To change an entire line, it is easier to use the Replace command than the Change command. Typing "R new line" will delete your current line, and replace it with the specified new line. Type:

>R must stop sometime. --->

Now "gathers no moss" has been deleted, and "must stop sometime" has taken its place.

If you wish to replace a single line with more than one line, type "R", followed by a carriage return. The Editor will put you into Input mode, and replace your current line with everything you type while in Input mode.

As an example, to put "Into each life a little rain must fall" on two lines, first locate the aphorism:

Into each life a little rain must fall.

Replace this with 2 lines: " but the second and the

>R --->
Input
Into each life
a little rain must fall.

Give two carriage returns to get out of Input and back in Edit.

3.7 Locating a String: The "T" and "N" Commands

If you wish to locate the first aphorism (A penny saved...), you can't use "L" or "P" because these commands only work downward from your current line. You could use the Up command, but since you know you're going to the top of the file, its easiest to use the Top command.

Type:

and you'll be at the top of the file.

If you want to see the next line, use the Next command by typing:

The following line is where the next change is to be made, so give the Next command again.

The line "is a penny earned" should now be on your console. Change this to "doesn't add up to much" using the Replace command:

Note: If you specify a number after "N" the Editor will execute "Next" that many times, but will only print the current line after completion.

3.8 Deleting Text: The "DE" Command of dayid and Deleting Text:

Locate, using any method you wish, "A stitch in time saves nine."

To get rid of this aphorism, use the Delete command. Make sure the line you wish deleted is above your ">" prompt, and type:

>DE --->

The aphorism is deleted and the Editor automatically closes up the blank line where the aphorism was.

If you wish to erase more text, simply specify the number of lines to delete, e.g., to delete 4 lines, type DE 4.

If you specify a string after DE, the editor will delete all lines down to, but not including, the first line containing that string. For example, if you later wish to delete all the aphorisms but the last one, you could go to the top of the file and type "DE /horse/".

3.9 Locating a String: The "B" Command

The Bottom command is another method of line location. To get to the bottom of the file, type:

>B --->
but you can't make it drink.

If you get a blank line instead of "but you can't make it drink," you added a few carriage returns at the end when originally inputting the file. If so, use the Up command (Up /but/), and the line will appear. Note that Up is like DElete, in that you can specify a string that you wish to reach instead of a number of times to repeat the command. This is true of Next and Print as well.

3.10 Adding Text: The "I" Command

To insert text after your current line, you must get into Input. Add to your file the line, "(It probably will, though)", by typing:

>I --->
INPUT
(It probably will, though).

Exit Input with two carriage returns. Remember to enter Input only when you are at the place in your file where you want text inserted.

3.11 Routine Editing Session Wrap up

- 1. Get out of Edit using the OUIT command.
- 2. Back-up the APHORISMS file.
- 3. Make a hard copy if a printer is available.

The end results of all your editing efforts should look like this:

A penny saved doesn't add up to much.

Into each life

a little rain must fall. Doy and teal and the amelia

Every Rolls Royce has a silver lining.

A rolling stone must stop sometime.

You can lead a horse to water, and the most on the same but you can't make it drink. (It probably will, though).

Any time you are in Edit, you can see what your file looks like by going to the top of the file (type "T", carriage return), and using the Print command to print out any number of lines. Typing "P *" will print the entire file, starting from your current line, on the console.

If your aphorisms file looks drastically different than the file listed above, change it, add to it, delete from it, etc., until it matches. See Section 3.12, Summary, for assistance.

3.12 Summary about and add

There is more to the RIO Text Editor, but the commands you know now are enough to start with. Here is a summary of what you have learned so far for easy reference:

| Command Abbreviation | What it Does |
|---|---|
| Replaces current line with the given line | R line |
| | file |
| C /old string/new string/ | Changes old string to man |
| Goes up apacified number of ad | Deletes current line |
| # 3D # Goes up to the specified string | Deletes specified number of lines |
| DE /string/ | Deletes up to, but not including, the first line with the specified string |
| I | Puts you in Input mode so that you may add text to your file. |
| L /string/ | Locates the first occurence (after your current line) of the specified string |
| N | Goes to the next line |
| N # | Locates the line the specified number of lines below your current point in the file |
| N /string/ | Locates the next line with the specified string (same as L /string/) |
| P | Prints your current line |
| P # | Prints the specified number of lines. |
| P /string/ | Prints all lines down through the specified string |
| QUIT | Gets you out of Edit |

Command Abbreviation

R line

T

U

U #

U /string/

What it Does

Row abanance and and another Puts you in Input mode and replaces the current work line with all text given while in Input

> Replaces current line with to mortod end of may age the given line

> > Gets you to the top of the file

Goes up one line

Goes up specified number of lines

Goes up to the specified string

DISK AND FILE KNOW-HOW

4.1 Formatting a New Disk

Every disk must be formatted before it can be used. In most cases, the disk to be formatted should have a silver tab placed over the notch (see Figure 1). Put a tab on the disk unless otherwise specified. This tab should cover the front and back of the notch. Its purpose is to let you write on the disk via the keyboard.

Put a System Disk into Drive 0 and bootstrap if the "%" prompt is not present.

If you are formatting a System Disk, type:

% FORMAT S --->

READY?Y

DRIVE: 2 DISK ID: Egret --->

(At this point, insert disk to be formatted into Drive 2) (Give it any name you like)

It is important to type "S" after "Format". This directs special System Disk information to its proper place on the disk.

When the computer issues a prompt, the disk is formatted. Now move all the programs from the Master System Disk in Drive O to the disk in Drive 2 by typing:

% MOVE P=& --->

When the red lights stop flashing and the prompt appears, you may remove and label the disk in Drive 2.

The procedure for formatting a User Disk is much the same, but even easier:

%FORMAT --->

Drive: 2

(Insert disk to be

formatted)

Disk ID: Heron ---> (Can be any name) Ready?Y

When the lights stop flashing and the prompt appears, you may remove the disk and label it. Don't type "S" after "Format" or give the Move Command, as this is only necessary for System Disks.

4.2 Copying a Disk

To make a copy of an entire disk, put the System Disk into Drive 0, bootstrap if the "%" is not present, and type:

% COPY.DISK --->
DRIVES READY?

Before continuing, remove the System Disk. Note that this is a safe exception to the rule about removing disks when the "%" prompt is not present. Put the disk you want to copy from into Drive 0, the disk you want to copy to into Drive 2, and continue:

Drives Ready?\\
VERIFICATION COMPLETE\\
\(^2\)

If any errors are listed on your console, it could mean that the disk you used in Drive 2 was bad, or that some minor electrical mishap occurred during copying. Try again.

4.3 Copying a File

The procedure for copying a file is the same as the procedure you used to create a back-up disk.

It is important to type "S" after "Format". This: sqyT special System Disk information to its proper place on

When the computer issues a prompt, the dest.

Remove the System Disk. Insert the disk with the file you wish to copy into Drive 2. Insert the disk you wish to copy the file onto, into Drive 0, and continue:

%X * S=2 D=0 Yourfile --->

FILENAME DRIVE YOURFILE 2

If you want to copy more than one file, you can list them all at once, leaving a space between each name:

%X * FILE 1 FILE 2 FILE 3 ---> ROYAL SET SERVE

If you leave out "S=2" and "D=0" in this command, the computer assumes your source is Drive 2 and destination is Drive 0.

4.4 Listing the Directory of Files

To see what files are on the disk in Drive 2, type:

%CAT D=2 --->

D=2 means "list the contents of Drive 2 only."

To see what files are on the disk in Drive 0, type:

%CAT D=0 P=& --->

P=& means "properties equals anything". The system will list everything on the disk in Drive O, including the special programs it contains if it is a System Disk.

4.5 Deleting a File

List the directory for Drive 2 and decide which file or files you want deleted. Now type "DELETE", followed by a list of the file(s) you want erased. For example, to delete three files called TRASH, JUNK and GARBAGE, you'd type:

%DELETE TRASH JUNK GARBAGE --->
DELETE 2/JUNK (Y/N/A/O)?

The computer's question is to make sure you don't accidently delete a desired file. You should give one of the listed answers -- Y,N,A or Q. "Y" means yes; "N" means no; "A" means all (delete all of the files listed and don't ask me again); and "Q" means quit (don't delete any files and stop the whole procedure). If you type "Y" or "N", the computer will ask you one by one whether you want a file deleted.

In the above example, RIO didn't forget the TRASH file. The computer deletes a series of files in the order that they were created, as opposed to the order in which you specify them. Therefore, JUNK must have been created before TRASH.

GOOD THINGS TO KNOW

5.1 RIO Editor Command Summary

Because the purpose of this document has been to give a quick overview of RIO, some unnecessary (but convenient) commands and options have been left out. The following is a complete list of RIO Editor Commands:

| Again | Macro |
|--------|---------|
| Bottom | Next |
| Brief | Print |
| Change | PUt |
| DElete | PUTD |
| Find | QUIT |
| GEt | Replace |
| Goto | Top |
| Input | Up |
| Join | Verify |
| LIneno | Window |
| Locate | Xecute |
| | |

The RIO Text Editor User's Manual, which describes these commands in detail, may be obtained from Zilog. See Section 7 for details.

5.2 Command Abbreviations

In the Command Summary above, some commands have only the first letter capitalized, while others have two or more. The capitalized letters indicate the minimum set of letters necessary to make the command, e.g., to use the Replace Command, only an "R" is necessary, while to use the DElete command, DE is necessary. If you wish, you can always use more letters.

While in Edit, (when the ">" prompt is present), you may give commands in lower case. This is not true outside of Edit mode; the "%" prompt requires capitalized commands.

5.3 * Option

The DElete, Next, Print, and Up commands can be issued with a number to specify the number of times to execute the command. If no number is specified, 1 is assumed. This is the "default value". An asterisk (*) can be used in place of a number to mean "as many times as possible". For example, "DE *" would delete

all the lines from the current one to the end of the file, and "P *" means "print all lines until the end of the file".

The * option also occurs in the change command. If you wish to change <u>all</u> remaining occurrences of String 1 to String 2, type "C /string 1/string 2/* *". See the RIO Text Editor User's Manual for more details of the change command, other commands with the * option, and other uses of *.

5.4 String Delimiters

This manual uses slash marks for string delimiters. Occasionally, however, you will need to change a string that contains a slash, e.g., change "in/out" to "out/in". Whenever a string is specified, the Editor considers the first non-numeric, non-blank character as the string delimiter. Therefore, to make the above change, pick one of your favorite symbols -- how about \$ -- and use it where you would normally use a slash:

C \$in/out\$out/in\$ ---> out/in

Note also that the last string delimiter in a command is optional. As an example, the above command can also be typed as "C \$in/out\$out/in". "L /horse/" can be written in short-cut form as "L /horse", etc.

all the lines from the current one to the end of the file, and "P "" means "print all lines until the end of the file".

The * option also occurs in the change command. If you wish to change all remaining occurrences of String 1 to String 2, type "C /string 1/string 2/* *". See the RIO Text Editor User's manual for more details of the change command, other commands with the * option, and other uses of *.

5.4 String Delimiters

This manual uses, slash marks for string delimiters. Occasionally, however, you will need to change a string that centains a slash, e.g., change "in/out" to "out/in". Whenever a string is specified, the Editor considers the first non-numeric, non-blank character as the string delimiter. Therefore, to make the above change, pick one of your favorite symbols -- how about first and use it where you would normally use a slash:

C \$im/out\$out/in\$ --->

Note also that the last string delimiter in a command is optional. As an example, the above command can also be typed as "G Sin/outSout/In". "L /horse/" can be written in short-out form as "G /horse", etc.

Z8000 PLZ/ASM Assembler User Guide

PREFACE

This manual describes how to use the Z8000 PLZ/ASM language translator (Z8000ASM). The Z8000 PLZ/ASM language is described in the Z8000/PLZ/ASM Assembly Language Programming Manual. Implementation dependent features are described in this document.

Z8000ASM depends on certain features of the RIO Operating System and runs on either an MCZ or ZDS system with $64 \mathrm{K}$ of memory. A complete description of the features in RIO can be found in the RIO Operating System User's Manual. Z8000ASM uses the stream I/O package to deal with files, and thus indirectly requires the memory management provided by RIO.

CONTENTS

| SECTION | 1 | INTRODUCTION | 332 |
|---------|---|---|-----|
| SECTION | 2 | Z8000ASM COMMAND LINE | 333 |
| SECTION | 3 | LISTING FORMAT | 335 |
| SECTION | 4 | LISTING CONTROL | 337 |
| SECTION | 5 | CONDITIONAL ASSEMBLY | 338 |
| SECTION | 6 | IMPLEMENTATION FEATURES AND LIMITATIONS | 339 |
| SECTION | 7 | OBJECT CODE TAGS | 341 |
| SECTION | 8 | PLZ/ASM ERROR MESSAGES | 344 |

INTRODUCTION

The RIO Editor should be used to create a Z8000 PLZ/ASM source program. The source file should end with the file name extension ".S" (upper or lower case).

The assembler creates a listing file (with default name of the source file except with the extension ".L" rather than ".S"), and an object file (with default extension ".OBJ"). In creating the object file, Z8000ASM uses a temporary intermediate file that is deleted when the assembly is finished. The listing file contains both the source with line numbers and any error numbers on a line following the corresponding source line. The list of error numbers in Section 7 can be used to determine the corresponding assembly error.

Z8000ASM COMMAND LINE

In the following description, the word "filename" is used to specify either a RIO file name or a RIO file name fully or partially qualified by a device name.

The assembler is invoked by the following general RIO command line:

Z8000ASM filename [options]

P

where filename has the extension ".S" and contains the source for a single Z8000 PLZ/ASM module. The extension is optional; if ".S" is omitted, the assembler appends an ".S" before attempting to open the file. The following options are valid and may appear in any order separated by delimiters such as blank or tab.

D=string Where "string" is at most 18 characters (not including delimiters such as blank, tab, return or semicolon) and is typically the date included in the heading of the listing.

I=filename Causes the intermediate file to be named "filename" instead of the default of a scratch file. The file may not be deleted upon completion.

L=filename Causes the listing file to be named "filename" instead of the default of the source file name with the extension ".L".

NOL Suppresses the generation of a list file.

O=filename

Causes the object file to be named "filename"

instead of the default of the source file name

with the extension ".OBJ". If no object is

desired, use \$NULL for "filename".

The Print option causes a copy of the listing to be routed to SYSLST and no compilation errors to be printed on the console. The default is that error numbers and the source line where the errors are detected are routed to CONOUT in addition to the listing file.

The assembler uses the following logical units for I/O:

| | E |
|---------------------------------------|--|
| Source file | |
| Listing file | |
| Object file To the Adam Parket Parket | |
| Intermediate file | |
| (02 (| Messages to the console Copy of listing if P option is specified Source file Listing file Object file Intermediate file |

LISTING FORMAT

The Assembler produces a listing of the source program along with generated object code. The various fields in the listing format are described below. Refer also to the sample listing in this section.

Heading

The first page heading contains the Assembler version number and column headings as explained below. In addition, the heading may contain a user-specified string that is usually the date of assembly (see Date option).

LOC

Location: This column contains the value of the reference counter for statements. Note that the counter starts at zero for each different section.

OBJ CODE

Object code: This column contains the value of generated object code. It is blank if a statement does not generate object code.

Each byte or word of object code is followed by either a single quote ('), an asterisk (*) or a blank. A single quote (') indicates that the value will be relocated. An asterisk (*) indicates that the value is dependent on an external symbol. A blank indicates that the value will not change. A value that is either to be relocated or that is dependent on an external is likely to be modified by either the Linker or loader. The value in the listing may be different than the value during program execution. Three dots "..." are used to indicate that the preceding byte, word or long word is repeated (only in data initialization).

STMT

Statement number: This column contains the sequence number of each source line.

SOURCE

The remainder of the line contains the source text.

! End of module !

50 END bubble sort

*EOF

LISTING CONTROL

The assembler directive \$LISTON is used to test the value of its operand and, depending on the result, inhibit listing of subsequent statements until another \$LISTON directive is encountered.

\$LISTON [constant expression]

The assembler evaluates the expression in the operand field of the \$LISTON directive. If the 32-bit result is true (non-zero), the \$LISTON directive is ignored and subsequent statements are listed normally. If the result is false (zero), the listing of subsequent statements is inhibited until another \$LISTON directive is encountered. If no argument is given, the directive is treated as if a true expression were specified.

Conditional listing blocks cannot be nested; the occurrence of a \$LISTON with no operand causes resumption of listing of subsequent statements.

EXAMPLES:

| Directive | Argument |
|-----------|---|
| \$LISTON | FLAG !these statements are listed only if the value of FLAG is not equal to zero! |

\$LISTON

CONDITIONAL ASSEMBLY

Conditional assembly allows the programmer to inhibit the assembly of portions of the source text provided certain conditions are satisfied. Conditional assembly is particularly useful when a program requires similar code sequences for slightly different applications.

Rather than generating a multitude of programs to handle each application, the application-dependent sections of code can be enclosed by the conditional pseudo-ops within a single program. By changing the values of several symbols used to control the conditional assembly, the user can generate different object modules from subsequent assemblies of the same source.

The assembler commands that allow conditional assembly are not currently implemented.

IMPLEMENTATION FEATURES AND LIMITATIONS

- Conditional assembly is not implemented in this release of Z8000ASM.
- 2. The Z8000 PLZ/ASM assembler uses the standard ASCII character set. Upper or lower case characters are recognized and treated as different characters; keywords will be recognized only if they are either all upper or all lower case (e.g., GLOBAL or global, but not Global). Hexadecimal numbers and special string characters can be either case (e.g., %Ab, 'lst line%R2nd line%r').
- 3. Source lines longer than 132 characters are accepted but only 132 characters are printed when printing out error messages. Note that comments and quoted strings may extend over an arbitrary number of lines, and caution should be exercised to avoid mismatched comment delimiters (!) or string delimiters (').
- 4. Strings may not be of zero length (e.g., '').
- 5. The symbol table currently contains room for approximately 400 identifiers, depending on the length of variable names and the complexity of declarations (e.g., ARRAYS, RECORDS). Identifiers may be of any length up to a maximum of 127 characters.
- 6. Constants are represented internally as 32-bit unsigned quantities and each operand in a constant expression is evaluated as though it was declared to be of type LONG. For example, 4/2 equals 2, but 4/-2 equals 0 since -2 is represented as a very large unsigned number. There is no overflow checking during evaluation of a constant expression. Because constants are represented as 32-bit values, only the first four characters in a character sequence, used as a constant, are meaningful (e.g., 'ABCD' = 'ABCDE').
- 7. The size of an array must be specified in a declaration. Note that [*] is not implemented.
- 8. When translating high-level constructs such as IF and DO statements, the assembler always generates JP instructions.

- 9. Because the PLZ/ASM assembler uses a recursive descent parsing method, the depth of nesting of constructs within a program is limited only by the available stack space. While the allocated stack space has been found to be adequate for all but the most contrived nesting, it is possible to modify the STACK_SIZE value of the Z8000ASM program using the RIO SET command to provide more (or less) stack space when the assembler is executed. Refer to the RIO Operating System User's Manual, for further details. Stack overflow checking is not done by the assembler.
- 10. Within CONSTANT, TYPE, or variable declarations, the assembler skips ahead until it finds the next keyword that will start a new statement (e.g., an opcode, IF, DO, EXIT, REPEAT or END). This skipping ahead may cause several assemblies before all errors are detected and removed.
- 11. The initialization symbols "?", "...", and nested
 constructors ('[' and ']') are not implemented.
- 12. Local labels (\$n) are not implemented.
- 13. Assembler directives must be typed in uppercase; e.g., \$section will not be recognized.

OBJECT CODE TAGS

This section includes a list of the object tags, their functions and the corresponding fields used by the Z8000 assembler. The tags are classified into three groups: control tags that are used to transfer control information; entry tags that define the code; and modifier tags that act as modifiers for the entry tags.

OBJECT CODE SYNTAX

NOTE: The object code format is still under development and is subject to change.

object_module => tagged_entry*

tagged_entry => control_entry | modified_entry

control entry => NOP

=> SEGMODULE size name

=> NONSEGMODULE size name

=> ENDMODULE

=> SECTION size name

=> GLOB sec loc attr typel type2 name

=> ABSGLOB sec loc attr type1 type 2 name

=> EXTERN attr typel type2 name

=> ENTRYPT sec loc
=> ABSENTRYPT sec loc

=> DEBUGSYMBOL bcount sec loc bval*

=> DEBUGFINO bcount bval*
=> MESSAGE bcount bval*

=> SETDATA sec => SETPROG sec => BEGSEC sec => LOCNT wval => ABSLOCNT wval

modified_entry => [REP bcount] [SHORT] [HIBYTE | LOBYTE]

[SEGMENT | OFFSET] [DISP wval] [REL sec | RELPROG | RELDATA] [SEQUENCE bcount] load_entry

=> LDBYTE bval load entry => LDWORD wval => LDLONG lval => EXREF ext => SECREF sec => SECADDR sec loc name => bcount byte*
bcount => byte
size => word attr _____ => byte ____ => byte typel => byte => byte type2 => byte sec loc => word bval => byte

=> long

=> word

lval

OBJECT CODE TAGS

CONTROL TAGS

| HEX | | |
|-------|--------------|---|
| VALUE | | |
| 00 | NOP | null operation |
| 01 | SEGMODULE | segmented module definition |
| 02 | NONSEGMODULE | nonsegmented module definition |
| 03 | ENDMODULE | end module |
| 04 | SECTION | section definition |
| 05 | GLOB | global symbol definition |
| 06 | ABSGLOB | global symbol definition with absolute offset |
| 07 | EXTERN | external symbol definition |
| 08 | ENTRYPT | entry point with relocatable offset |
| 09 | ABSENTRYPT | entry point with absolute offset |
| OA | DEBUGSYMBOL | debug symbol |
| 0B | DEBUGINFO | debug information |
| 0C | MESSAGE | variable length message |
| 0D | SETDATA | set current data section |
| 0E | SETPROG | set current program section |
| OF | BEGSEC | begin section |
| 10 | LOCNT | relocatable program counter |
| 11 | ABSLOCNT | absolute program counter |
| | | |

ENTRY TAGS

| VALUE 20 | LDBYTE | load byte value | |
|-------------|---------|--------------------|--|
| | | | |
| 21 | LDWORD | load word value | |
| 22 | LDLONG | load long value | |
| 23 | EXREF | external reference | |
| 24 | SECREF | section reference | |
| 25 | SECADDR | section address | |
| | | | |

MODIFIER TAGS

| HEX | | |
|------|-----------|--|
| VALU | E | |
| 40 | REP | repeat badaegka '17' |
| 41 | SEQUENCE | sequence |
| 42 | REL | relocatable |
| 43 | RELDATA | relocatable with respect to current data area |
| 44 | RELPROG | relocatable with respect to current program area |
| 45 | DISP | displacement |
| 46 | *LOBYTE | low order byte of |
| 47 | *HIBYTE | high order byte of |
| 48 | **SHORT | short segment address |
| 49 | **OFFSET | offset of |
| 4A | **SEGMENT | segment of |
| | | |

^{*} Z8 **Z8000

PLZ/ASM ERROR MESSAGES

| ERROR | EXPLANATION |
|----------------------------------|---|
| | Warnings |
| 1 2 3 4 5 | Missing delimiter between tokens Array of zero elements No fields in record declaration Mismatched procedure names Mismatched module names |
| | Token Errors |
| 10 11 12 13 14 15 | Decimal number too large Invalid operator Invalid special character after '% Invalid hexadecimal digit Character sequence of zero length Invalid character Hexadecimal number too large |
| | DO Loop Errors |
| 20 21 22 23 24 | Unmatched 'OD' 'OD' expected Invalid repeat statement Invalid exit statement Invalid 'FROM' label |
| | IF Statement Errors |
| 30 31 32 | Unmatched 'FI' 'FI' expected 'THEN' or 'CASE' expected |
| | Symbols Expected |
| 40 41 42 43 | ')' expected '(' expected ']' expected '[' expected ':=' expected |

| ERROR | EXPLANATION MOLITAMATIKA |
|--|---|
| | Undefined Names |
| 50 51 | Undefined identifier Undefined procedure name |
| | Declaration Errors |
| 60 61 62 63 64 65 66 | Type identifier expected Invalid module declaration Invalid declaration class Invalid use of array [*] declaration Uninitialized array [*] declaration Invalid dimension size Invalid array component type Invalid record field declaration |
| | Procedure Declaration Errors |
| 70 71 72 | Invalid procedure declaration 'ENTRY' expected Procedure name expected after 'END' |
| | Initialization Errors |
| 80 81 82 83 | Invalid initial value Too many initialization elements for declared variables Invalid initialization Array [*] given single non-character_sequence initializer |
| | Special Errors |
| 90 91 92 93 94 95 | Invalid statement Invalid instruction Invalid operand Operand too large Relative address out of range ":" expected Multiple declaration of identifier |
| | Invalid Variables |
| 100 101 102 103 104 | Invalid variable Invalid operand for '#' or 'SIZEOF' Invalid field name Subscripting of non-array variable Invalid use of '.' |

| ERROR | EXPLANATION MOTTAMATEMENT |
|--|---|
| | Expression Errors |
| 110 111 112 113 114 115 | Invalid arithmetic expression Invalid conditional expression Invalid constant expression Invalid select expression Invalid index expression Invalid expression in assignment |
| | Constant Out of Bounds |
| 120 121 122 | Constant too large for 8 bits Constant too large for 16 bits Constant array index out of bounds |
| | Type Incompatibility |
| 140 | Character_sequence initializer used with array [*] declaration where component's base type is not 8 bits File Errors |
| 198 199 | EOF expected Unexpected EOF encountered in source |
| | Implementation Restrictions |
| 229 230 231 234 235 236 254 255 | Unimplemented feature Character sequence or identifier too long Compiler Symbol table overflow Too many initialization values Compiler stack overflow Operand too complicated Intermediate file error BUG |
| | |

Z8000 Linker User Guide

PREFACE

This manual describes how to use the Zilog Linker in conjunction with the Z8, the segmented Z8000, the nonsegmented Z8000 and the Z-80 processors. The Z8000 PLZ/ASM language translator is described in the Z8000 PLZ/ASM Assembler User Guide and the Z8000 PLZ/ASM language is described in the Z8000 PLZ/ASM Assembly Language Programming Manual.

The Linker depends on certain features of the RIO Operating System and can run on either an MCZ or ZDS system with 64K of memory. A complete description of RIO features can be found in the RIO Operating Systems User's Manual.

CONTENTS

| SECTION 1 | OVERVI | EW | 350 |
|------------|---------|---|--------------------------|
| | | ntroductionunctions and Features | 350 351 |
| SECTION 2 | LOAD MO | ODULE 1 Description of deplete the second of the seco | 352 |
| SECTION 3 | SECTION | NS | 353 |
| SECTION 4 | LINKER | OPERATION | 354 |
| | 4.2 Co | xecution | 354 355 356 |
| | 4 4 4 | 3.1 Link Map | 356 356 357 357 |
| | | ogical Unit Definitions | 357 |
| SECTION 5 | | PERATION EXAMPLES | |
| | | | 358 |
| SECTION 6 | | MESSAGES | 361 |
| APPENDIX A | COMMANI | D LINE SYNTAX | 365 |
| APPENDIX B | OBJECT | CODE SYNTAX | 366 |
| ADDENDIA C | CAMDIF | I TNE MAD | 770 |

OVERVIEW

1.1 Introduction

The Linker is a general-purpose programming tool intended for use with several Zilog processors: in particular, the Z8, the segmented Z8000, the nonsegmented Z8000, and the Z-80. The Linker is general enough to handle the differing architectures of these processors, yet includes enough capabilities to allow the user to configure his program as desired.

The Linker accepts several object modules produced by an assembler or compiler and combines them into a single load module. The output of an assembler or compiler is an object module containing enough information to allow the Linker to assign that module to a memory area. Because many modules may be loaded together to form a complete program, inter-module communication is needed. For example, one module may contain a call to a procedure that was assembled as part of another module and may be located in some arbitrary part of memory. Therefore, the assembler or compiler must provide information in the object module that allows the Linker to link inter-module references.

There are several major advantages to using separately assembled or compiled modules as compared to a single program module:

- 1) Assignment of modules to memory areas can be handled by the Linker rather than requiring the programmer to assign fixed absolute locations via the \$ABS directive; thus, modules can be relocated without requiring re-assembly.
- 2) If errors are found in one module, only that one module needs to be re-assembled and then re-linked with the other modules.
- 3) Programs can be structured into independent modules, coded separately and assembled, even though the other modules may not yet exist. This separation of functionality also enhances program maintenance and verification.
- 4) Libraries of commonly-used modules can be built, then linked with programs, without requiring their re-assembly.

Addresses within a program module may be either relocatable or absolute. The notion of "absolute" assumes that there is a one-to-one correspondence between the assigned address of an instruction within a module during assembly and the memory address of that instruction during program execution. Thus, a module whose origin is "absolute 0" will have its first instruction located at memory location 0.

"Relocation" refers to the ability to assign a program module and its data to a particular memory area at a time after the assembly or compilation process. The Linker program is used to combine and relocate several object modules into a single load module, while a loader program is used to relocate and load a single load module into specific memory addresses.

The notion of "relocatable" assumes that the assigned address of an instruction within a module during assembly is a relative offset from an origin that is assigned an absolute address by the Linker and loader. During program execution, the instruction is located at the memory location specified by the assigned origin plus the relative offset. Thus, a relocatable module whose first instruction (which is located at "relocatable 0") will have that instruction located at the memory location that is the assigned origin of the module as determined by the Linker and loader.

1.2 Functions and Features

The Linker performs the following functions: 1) It combines separately assembled modules, resolving external references; 2) it provides an optional link map showing how modules and sections have been relocated and where global variables are located; 3) it reports errors in the linking process to the user via the console and the link map.

Linker object module input and load module output have the same format. This allows the Linker to be used "incrementally", whereby a group of modules can be combined and the resulting load module can be used as an object module; i.e., as input to another execution of the Linker.

One of the important features of the Linker is the versatility it allows in arranging program and data. The PLZ/ASM assembly language programmer is given the option of partitioning code into named "sections", discussed in detail in Section 3.

SECTION 2 TO BE THE BEST OF THE SECTION 2

One-Lo-one correspondence AJUDOM CAOL ne assigned address of an Lustruction within a module ouring assembly and the memory during program execution. Thus, a

The output from the Linker, called the load module, will be processed by a loader when the user wishes to execute the program. The form of the file is similar to the object files generated by the assembler. One advantage to keeping the structure the same is that simple programs (i.e., those without external references) may be processed by the loader without first being processed by the Linker. Moreover, keeping the file structure consistent between the Linker and assembler aids in the portability of the output files between these translators.

The structure of the load module is easily extendable. Future versions of the Linker can include more complex processing without making previously defined load modules invalid. Note that the load module is, in fact, an object module. The output from the Linker may be considered as any other object module and may be further linked in subsequent usage of the Linker.

The load module consists of control information, called tags, followed by the actual values to be loaded. The tags indicate relocatability, address structure, messages, etc. This file is processed sequentially. For a listing of these tags, see Appendix B. Note that the format is still under development and subject to change.

SECTIONS

The PROCEDURE and MODULE constructs are the primary means of organizing code. In addition, a PLZ/ASM assembler directive allows the programmer to partition pieces of code into named sections. A group of procedures or a certain set of data may belong to a single, user-named section. By default, an assembler or compiler separates a module into the sections "MODULENAME P" (for PROGRAM) and "MODULENAME D" (for DATA). Such partitioning can aid the programmer in debugging and in understanding complex programs, and may be used to control the final mapping of program and data into memory. This user-control is useful when mapping code into different segments in the segmented Z8000, or when it is desirable to physically separate the code for the procedures from the data in a system where read-only memory is used for the procedures and read/write memory is used for the data.

The Linker preserves the notion of sections in the load module. Each unique section will be mapped into a different "area" of memory. For the purposes of the segmented Z8000, an area may be a segment. An area in the nonsegmented Z8000 or the Z-80 is a contiguous portion of memory. In the Z8, an area is a contiguous set of memory locations in one of the three available address spaces: program, data or register. The user may combine sections named in his code into a single section at link time. For example, sections from different modules that logically belong together, such as data sections, can be loaded into memory in an arbitrary segment or at an arbitrary memory location, but remain separated from other such sections.

The concept of sections encourages the user to organize his program into areas and not depend on absolute memory locations. This allows for greater portability of code. It also encourages development of code that does not depend on being loaded into absolute memory locations or expect other code to reside at absolute memory locations.

LINKER OPERATION

4.1 Execution ag edd era adourdance alugom bas asuggoong edt

The Linker is executed by typing the following command:

LINKZ8000 command line

The command line, explained in detail in Section 4.2, consists of the names of the input files to be processed and the Linker options. Any errors found in the command line result in an immediate abort of the Linker. The command line syntax is detailed in Appendix A. The user may cause the Linker to abort at any time by hitting the ESCAPE key.

The Linker produces an output file, called the load module. This process occurs in two passes. During the first pass, the modules are broken up into their sections, and sections with the same name are grouped together. This grouping may involve sections in distinct modules. The Linker opens a temporary file to contain this reorganized code. A directory containing the global symbols and their locations is also built during the first pass. Any missing input files or input files containing erroneous object code will suppress the creation of the load module. A description of the object code format appears in Appendix B.

The second pass of the Linker organizes the sections as specified by the user and resolves external references. Any unsatisfied externals sends a warning to the user. Currently, an unsatisfied external that is referenced causes a default value to be placed in the object module for that address. In a subsequent release, any unsatisfied external references will remain in the load module so that the resulting "partially linked" load module can be used in another linkage.

All errors or warnings occurring during the link process cause a message to be sent to the console. If a link map is requested, all error messages will appear there as well. All possible error messages and explanations are listed in Section 6. If no errors occur during the link process, the load module is created. If requested, the link map will also be made. An example of a link map is shown in Appendix C.

4.2 Command Line

The notation and options that can appear in the command line of the Linker are listed below. The ordering or placement of the Linker options within the command line is free-format. Each item in the command line must be separated using one or more spaces, tabs, commas or linefeeds. The command line is terminated by a carriage return or a semicolon.

All object files to be used for linking must be specified by name in the command line. The order in which they are specified indicates the order in which they are processed by the Linker. The file names need not appear before the options or section information; i.e., they may be intermixed. Object file names are assumed to have the extension ".OBJ" (or ".obj" if the first character of the filename is lowercase). If this extension is not specified in the command line, it will be added to the file name by the Linker.

EXAMPLE:

LINKZ8000 MAINPROGRAM PASSONE PASSTWO.OBJ

Sections may be logically grouped using parentheses. For user-named sections PROCESS_ONE, DATA_ONE and GENERATE_ONE, a grouping would look like:

EXAMPLE:

(PROCESS_ONE DATA_ONE GENERATE ONE)

Groups of sections may be named so that they may be used as a single section in later linkages.

EXAMPLE:

ALL_ONE = (PROCESS_ONE DATA_ONE GENERATE ONE)

To help the programmer organize groupings easily, an asterisk (*) may be used in specifying section names. One or more asterisks may be placed within the specification of any section name. The Linker will look for section names that exactly match the specified portion of the name and arbitrarily match the star. For example, "*" would match any section name while "*DATA" matches MYDATA, YOURDATA, HERDATA and OURDATA. Using the star character, the above example may be shortened to:

$$ALL_ONE = (*_ONE)$$

There may be many groups in any one link operation. The order of the groups as specified corresponds to their order in memory. The order of the section names within a group denotes the order of the code within that area of memory. Note that sections in different modules with identical names are treated as one section.

4.3 Options

The following constructs are the options the user may specify anywhere in the link command line. Generally, they are a command key word, or an abbreviated form, followed by an "=" character and then a file or symbol name. The key word may be upper or lower case, and is distinguished only by its first character. The file name may be replaced by a "*". This signifies that the default file name should be used for that option. 4.3.1 Link Map order the before the option of the file names need not appear the file names need not appear before the option

LINKMAP = default_or_file

The default for the link process does not create a link map. A link map may be obtained by specifying a link map file name. A default name for the link map may be specified by using "*" in place of a file name. The default is the first object file name in the command line, with the extension ".MAP" appended instead of the ".OBJ" extension. This listing includes the modules used, section content and an alphabetized listing of global symbols. An example of a link map appears in Appendix C.

EXAMPLE .

L = mylink.map

4.3.2 Entry

ENTRY = name

An entry point may be specified in the link command line. The entry point is the location where execution begins. The default entry point is the first entry symbol encountered in the modules processed. Specifying an asterisk in this option is a null operation.

EXAMPLE:

E = myentrypt

4.3.3 Load Module Name

NAME = default or file

The default load module name is the first file name specified in the command line without the ".OBJ" extension. The user may specify any file name by using the naming option.

EXAMPLE :

N = MYLOADMODULE

4.3.4 Temporary File Name

TEMPORARYFILE = default or file

The temporary file created by the Linker during the first pass of operation can be named by the user. Naming this file saves the intermediate file for the user. The default name of this file (specified by using "*") is the name of the first object file in the command line, with the extension ".T" appended instead of the ".OBJ" extension. If this option is not specified, the temporary file will be created as a scratch file, and will be deleted after completion of the link process.

EXAMPLE :

T = MYTEMPFILE

4.4 Logical Unit Definitions

The Linker uses the following logical I/O units:

- 2 Error messages
 - 4 Object module files
 - 5 Temporary file
 - 6 Load module file
 - 7 Link map file Table 1997 The Table

LINK OPERATION EXAMPLES

This example shows how to link the following modules:

| Module Name | File Name | Sections in Module |
|------------------|------------|-------------------------------|
| A | A.FILE.OBJ | A_D emporacy File Tene |
| | | CONTROL |
| | | INTERFACE |
| Maming this Bide | B.FILE.OBJ | temporary file creates |
| | | not elli edellement e |
| ton Cal noligo | C.FILE.OBJ | SORT |
| | | MERGE |
| | | SORT_D |
| | | MERGE_D |
| D | D.FILE.OBJ | amo D_D wo iinU labi |
| | | Q_Q Linker uses the follow |

The link operation to put data in one area and separate CONTROL code from the program code in the rest of modules is as follows:

NOTE: Each section name not specified is mapped by default into a separate area.

The configuration of the load module would look like:

| The state of the s | |
|--|---------------------|
| A_D | NKZ8000 A.FILE B.FI |
| B_D | 11.5 duli.A goodaan |
| SORT_D | SECTION #1 |
| MERGE_D | CONTROL |
| D_D | INTERFACE |
| CONTROL | SECTION #2 |
| MERGE | a_8 · · · |
| INTERFACE | SECTION #3 |
| SORT | MERGE |
| B_P | SECTION #4 |
| D_P | SECTION #5 |
| | |

For the segmented Z8000, each section could be mapped to a segment at load time; for the Z-80, nonsegmented Z8000 or the Z8, the load module would be read into contiguous memory locations.

Compare the organization of the above load module with the following link option, where no groupings have been specified:

LINKZ8000 A.FILE B.FILE C.FILE D.FILE

| A_D | SECTION #1 |
|--|--------------|
| CONTROL | SECTION #2 |
| INTERFACE | SECTION #3 |
| C # MCB_P 33 | SECTION #4 |
| B_D | •MERGE |
| SORT | -INTERFACE |
| MERGE | -SORT |
| SORT_D | e B. P |
| MERGE_D | 9_Q. |
| D_P | |
| D_Deanon | SECTION #11 |
| Lancius de la constitución de la | netti Diteri |

ERROR MESSAGES

All errors that occur in the Linker cause a message to be sent to the console. If requested, the message is printed in the link map file as well. There are three classes of error messages:

Class I - Fatal

Class II - Terminal (non-fatal)

Class III - Warning

Error messages and their meanings are listed below.

CLASS I Errors

The following errors cause an immediate abort of the Linker:

| MECCACE | MEANING |
|---------|----------|
| MESSAGE | PIEANING |
| | |

ILLEGAL CHAR IN COMMAND LINE : <c> A character not recognized by the Linker was found in the command line.

ERROR IN GROUPING FORMAT (symbol) An error was made in the command line when specify-ing a grouping. The symbol causing the error is displayed.

OPTION NOT RECOGNIZED IN COMMAND LINE <c>

Invalid option. The first letter of the option is displayed.

NAME EXCEEDS MAXIMUM : <name> A filename exceeds the allowable length.

UNABLE TO OPEN TEMPORARY FILE: <name>

Unable to open temporary file called "name".

UNABLE TO OPEN OBJECT FILE : <name> Unable to open output object file called "name".

MESSAGE MEANING

UNABLE TO OPEN INPUT FILE: <name> Unable to open input file

called "name".

UNABLE TO OPEN LINK MAP FILE: <name>

fods as bestging as epassem eds have Unable to open link map file called "name".

WRITE FILE ERROR, ERR CODE : <hexcode>

A system error occurred while writing to a file. System error code is displayed.

READ FILE ERROR, ERR CODE : <hexcode>

A system error occurred while reading a file. System error code is displayed.

SEEK ERROR, ERR CODE : <hexcode>

An error occurred during attempt to seek to a certain besinpoper don responsed A co., position in a file. System error code is displayed.

UNABLE TO CLOSE FILE <name>, ERR CODE : <hexcode>

A system error occurred while trying to close file "name". The system error code is displayed.

SYMBOL TABLE OVERFLOWED

The space allocated for symbol information has been filled. How quickly the table is filled depends on the quantity and length of the global symbols in the modules linked.

GENERAL TABLE OVERFLOWED

The space allocated for general information about sections, file names, etc., has been filled. How quickly the table is filled depends Joseph Jugue made of seldand vansar on the number of sections in the modules.

MESSAGE

MEANING

BAD CALL TO ERROR ROUTINE <err>

The internal Linker error routine received a call it could not recognize.

FEATURE NOT IMPLEMENTED <c>

LEMENTED <c>
An option was specified in the command line that has not been implemented yet. The first character of the option is shown.

CLASS II Errors

The following errors cause the Linker to abort after termination of the first pass:

MESSAGE

MEANING

MULTIPLE DECLARATION OF GLOBAL: <name> <module1> <module2>

Global "name" defined in "modulel" was previously defined in "module2"

NONSEG TAG IN SEG MODULE

A tag relating to nonsegmented code was found in a segmented module.

SEG TAG IN NONSEG MODULE

A tag relating to segmented code found in a nonsegmented module.

SEG AND NONSEG MODULES MIXED

Segmented and nonsegmented modules have been encountered during the link.

OBJECT CODE CONTAINS ERROR (name) (id)

The object code in file "name" is erroneous; "id" is error code.

BYTE OVERFLOWED

A relocatable byte value became too large for eight bits.

SECTION TOO LARGE <name>

The size of section "name" has exceeded the maximum size of 65,536 bytes.

CLASS III Errors

The following errors cause a warning to be sent to the console:

MESSAGE MEANING

GLOBAL TYPE MISMATCH : <name> <module1> <module2>

A type mismatch for "name" occurred in a cross module reference between "module1" and "module2".

UNDEFINED GLOBAL : <name> <module>

External reference "name" declared in "module" was not defined.

REFERENCE TO UNDEFINED GLOBAL: <name> <module>

A reference to undefined global "name" was made in "module".

NO ENTRY POINT FOUND

There was no entry point found in the modules processed.

APPENDIX A

COMMAND LINE SYNTAX

The syntax for the linker command line is as follows:

| LINKZ8000 | option | s* terminator |
|-----------------|----------------------|--|
| options | <pre></pre> | command grouping file_list |
| Dommand Same | => => => => | <pre>E[NTRY] '=' name L[INKMAP] '=' default_or_file N[AME] '=' default_or_file T[EMPORARYFILE] '=' default_or_file</pre> |
| grouping | => | [name '='] group_list |
| group_list | /d=>01 | '(' section_list ')' |
| section_list | => | <pre>name [separator section_list]</pre> |
| file_list | => | file_name [separator file_list] |
| default_or_file | => => | file_name_tsvw_TMOO1884 <= |
| name STYSOI | => | PLZ-defined section or variable name |
| file_name | => | system-defined file name |
| separator | => | ' ' ',' LF TAB |
| terminator | => | ';' CR favw dscwdd c= |

APPENDIX B

OBJECT CODE SYNTAX

NOTE: The object code format is still under development and is subject to change.

object module => tagged entry*

tagged entry => control entry | modified entry

control entry => NOP

=> SEGMODULE size name
=> NONSEGMODULE size name

=> ENDMODULE

=> SECTION size name

=> GLOB sec loc attr typel type2 name
=> ABSGLOB sec loc attr typel type2 name

=> EXTERN attr type1 type2 name

=> ENTRYPT sec loc => ABSENTRYPT sec loc

=> DEBUGSYMBOL bcount sec loc bval*

=> DEBUGINFO bcount bval* => MESSAGE bcount bval*

=> SETDATA sec => SETPROG sec => BEGSEC sec => LOCNT wval => ABSLOCNT wval

load entry

=> LDBYTE bval => LDWORD wval => LDLONG lval => EXREF ext => SECREF sec => SECADDR sec loc

| name | => | bcount byte* |
|--------|----|-----------------------------------|
| bcount | => | byte |
| size | => | word |
| attr | => | byte |
| typel | => | byte |
| type2 | => | byte |
| sec | => | byte |
| loc | => | word |
| bval | => | byte substitute to the design and |
| wval | => | word slubos ba |
| lval | => | long |
| ext | => | word |
| | | |

OBJECT CODE TAGS de SMEAT

CONTROL TAGS

| HEX | | |
|-------------|--------------|---|
| VALUE 00 | NOP | null operation |
| 01 | SEGMODULE | segmented module definition |
| 02 | NONSEGMODULE | nonsegmented module definition |
| 03 | ENDMODULE | end module |
| 04 | SECTION | section definition |
| 05 | GLOB | global symbol definition |
| 06 | ABSGLOB | global symbol definition with absolute offset |
| 07 | EXTERN | external symbol definition |
| 08 | ENTRYPT | entry point with relocatable offset |
| 09 | ABSENTRYPT | entry point with absolute offset |
| 0A | DEBUGSYMBOL | debug symbol |
| 0B | DEBUGINFO | debug information |
| 0C | MESSAGE | variable length message |
| 0D | SETDATA | set current data section |
| 0E | SETPROG | set current program section |
| OF | BEGSEC | begin section |
| 10 | LOCNT | relocatable program counter |
| 11 | ABSLOCNT | absolute program counter |

ENTRY TAGS

| HEX | | |
|-------|---------|--------------------|
| VALUE | | |
| 20 | LDBYTE | load byte value |
| 21 | LDWORD | load word value |
| 22 | LDLONG | load long value |
| 23 | EXREF | external reference |
| 24 | SECREF | section reference |
| 25 | SECADDR | section address |

MODIFIER TAGS

| HEX VALUE | | |
|--------------|-----------------|--|
| 40 | REP SEQUENCE | repeat |
| 42 | REL | sequence relocatable |
| 43 | RELDATA | relocatable with respect to current data area |
| 44 | RELPROG | relocatable with respect to current program area |
| 45 | DISP | displacement |
| 46 | *LOBYTE | low order byte of |
| 47 | *HIBYTE | high order byte of |
| 48 | **SHORT | short segment address |
| 49 | **OFFSET | offset of |
| 4A | **SEGMENT | segment of |
| | | |

* Z8 **Z8000

OBJECT CODE TAG FIELDS

CONTROL TAGS

| NOP SEGMODULE NONSEGMODULE ENDMODULE SECTION GLOB ABSGLOB EXTERN ENTRYPT ABSENTRYPT DEBUGSYMBOL DEBUGINFO | size size size sec# sec# attr sec# bcount | name name loc loc typel loc loc sec# | attr attr type2 | typel typel name | type2 type2 | name |
|---|--|--------------------------------------|-----------------------|------------------------|----------------|------|
| DEBUGINFO MESSAGE | bcount | sec# | 100 | | | |
| SETDATA | bcount sec# | e sect | | | | |
| SETPROG BEGSEC | sec# | | | | | |
| LOCNT ABSLOCNT | wval | | | | | |
| | ol sdr | | | | | |

ENTRY TAGS

| ENTRY TAGS | | |
|---|-------------------------------------|--|
| LDBYTE | bval | FIELD SIZES |
| LDWORD | wval | name : length byte, character string |
| LDLONG EXREF SECREF SECADDR | lval ext# sec# sec# offset | size : word sec# : byte loc : word attr : byte typel : byte |
| MODIFIER TAGS | | type2 : byte bval : byte wval : word |
| REP SEQUENCE REL RELDATA RELPROG | bcount bcount sec# | <pre>lval : long bcount : byte ext# : word offset : word</pre> |
| DISP *LOBYTE *HIBYTE **SHORT **OFFSET **SEGMENT | offset | |

^{*28} modules only **28001 modules only

APPENDIX C

SAMPLE LINK MAP

The sample link map contained herein consists of four major areas. The first is a copy of the command line as received by the linker. The second area is a message area. It contains any error messages or warnings that occurred during the link process. In this example, "NO ENTRY POINT FOUND" is a warning to the user. This area also includes a message indicating the size and type of module produced. The third area is a mapping of sections found in the input modules to the sections found in the output module. The size and location of each input section is listed in the section group it is found. The section in which the module was found is also listed. For example, section "MERGE D" from module "C" is located in section "DATA" at 2C4. Its size is 18 bytes. The last area is an alphabetized listing of all globals found in the input modules. For each global, the location, the output section it is located in, and the module in which it was found are listed.

LINKER 0.1

COMMAND LINE EMAN DOD TOSE SINGON

LINKZ8000 A.FILE B.FILE C.FILE D.FILE DATA=(* D) (CONTROL)

(MERGE INTERFACE SORT) RESTP_PROGRAM=(* P) L=*

N=MYLINKEDCODE

| NO ENTRY POINT FOUND | |
|------------------------------|------|
| NON-SEGMENTED MODULE, SIZE : | 4E2 |
| O SEE NOT | |
| MODULE NAMES | SIZE |
| 4 0 208000399 | A |
| U A | EE |
| TABABBINA XIS 0 | 262 |
| 2 C 337 HT 0 E | D4 |
| 3 D VASIA ULL | DE |

SECTION NAMES

| SIZE | LOC | SECTION | MODULE |
|------------------------------------|------------------------------|--|-----------------------|
| 332 86 1FA 44 18 56 | 0 86 280 2C4 2DC | SECTION DATA A_D B_D SORT_D MERGE_D D_D | A B C C D |
| 40 30 10 | 0 30 | SECTION CONTROL CONTROL | A C_ |
| A0 38 38 30 | 0 38 70 | SECTION MERGE INTERFACE SORT | C_ A C_ |
| D0 68 68 | 0 68 | SECTION RESTP_PROGRAM B_P D_P | B D |

GLOBAL SYMBOLS

| SECT | LOC | NAME | MODULE | SECT | LOC | NAME SMILL ON | MODULE |
|---|---|---|----------|---|--|---|--|
| 0 0 0 0 3 0 3 0 0 3 0 | 2DA 4 8A 68 2CA 30 2C4 6 0 0 2C0 280 | ABSOLUTELY COUNTER DELTA EIGHT FIVE FOUR HEADER LIST ONE REGPAIR SEARCH SORTA | C_ABDODE | 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 | 86 2D8 32C 2DC 2C2 88 32E 98 8C 2 | BETA DEFINED DID EXTRA FOUND GAMMA KNOT NINE_AND_LAST PSI REGQUAD SIX_AND_SEVEN THREE | B C D D C B D D B A C B |
| 1 | 30 | TWO | A | 0 | 330 | VERY | D |

MacP User Guide

CONTENTS

| SECTION | 1 | MacP DESCRIPTION 378 | 5 |
|----------|---|---|-----|
| SECTION | 2 | MACRO DEFINITION AND MACRO CALL PARSING 376 | 6 |
| | | 2.1 Input Stream | 78 |
| SECTION | 3 | EXPANSION PROCESS 388 | 5 |
| | | 3.1 Body of a Macro: Part I | 6 |
| | | Body 38 | 8 |
| SECTION | 4 | ARITHMETIC: DATA STORAGE AND HANDLING 39 | 0 |
| SECTION | 5 | COMMANDS 39 | 1 |
| | | 5.1 Conditional Expansion of Macros | 2 2 |
| SECTION | 6 | MacP OPERATION 39 | 4 |
| APPENDIX | A | PARSING COMPLEXITIES 39 | 6 |
| APPENDIX | В | ERRORS 40 | 0 |
| APPENDIX | С | I/O UNITS 40 | 13 |
| ADDENDIV | D | OUTDIT FORMAT |)4 |

MacP DESCRIPTION

MacP is a macro processor system designed specifically as a program preprocessing tool.

MacP processing begins with a sequential scan of an input stream. This input stream consists of a sequence of tokens (defined in Section 2). If the current token denotes a special action, that action is performed, else the token is sent to the output stream (which is another token sequence). There are two kinds of special actions that a token might trigger: macro calls and commands.

A macro call is a sequence of tokens; the first token is a macro name, and the whole sequence matches the corresponding macro template.

Matching the macro call to the template is called "parsing" the macro call. The parsing process uncovers the syntactic structure of the macro call, thereby determining the parameters for the macro call and the kind of repetitions or choice of options that are implied in the macro call.

After parsing, macro expansion occurs. MacP scans the macro body associated with the invoked macro as if it were part of the input (inserted in place of the macro call). The macro body may refer to the parameters discovered during parsing. Once macro expansion is done (the macro body may include macro calls, so the parsing-expansion process may have been executed recursively), the scanning of the input stream continues after the initial macro call.

A command is a request for service from MacP. Commands have the same structure as macro calls (their syntax is defined by means of a macro template, and the parsing process is the same) but instead of an expansion, their effect is to alter the internal status of MacP.

Commands allow the user to:

- Define new macros (&let)
- Control the scanning (&if)
- Control the input stream (&include)
- Organize macros in libraries and selectively choose the scope of a macro definition (&create, &activate, &libraries)

MACRO DEFINITION AND MACRO CALL PARSING

2.1 Input Stream Dempiseb medaya reseasong orosm s si 905M

The input stream is a set of tokens; token structure is defined by a set of token rules known to the MacP text scanner. The token rules state:

- which byte values represent valid characters (for instance, the ASCII character set)
- which characters have special meaning (for instance, every command name starts with a special character, the marker)
- how characters form tokens
- what to do with the string of separators between tokens (including comments)

The standard set of token rules is similar to the PLZ token definition. This means that tokens are basically identifiers, numbers, special symbols and strings.

| marker | After parsing, macro expansion & c= |
|--|--|
| letter | => 'A' 'B' 'Z' => 'a' 'b' 'z' marker |
| digit | => '0' '1' '2' '3' '4' => '5' '6' '7' '8' '9' |
| hex_digit smmoD .gosM moz eniled al xednys | => digit => 'A' 'B' 'C' 'D' 'E' 'F' => 'a' 'b' 'c' 'd' 'e' 'f' |
| delimiter | <pre>=> ',' ';' space => tab formfeed linefeed => carriage_return</pre> |
| special_symbol | => any_other_ASCII_character |
| identifier | => letter => identifier letter => identifier '' => identifier digit |
| number | => digit ==> number dig |

hex number => % hex digit

string => '''sequence of characters'''

token => identifier

=> number => hex number

=> string

=> special symbol

The construct

'!' any sequence of symbols not containing ! '!'

may be inserted anywhere that a delimiter may appear. It is called a comment and is treated as a single delimiter.

There is an important extension: a character sequence enclosed in double quotes (") is treated as a single token that does not match any other token. Thus enclosing a sequence in double quotes ensures that no macro expansion will alter the enclosed string.

The input stream is formed by a sequence of files. The sequencing is controlled by the &include command.

2.2 Macro Definition

A macro is defined by the '&let' command:

macro_name => token masside dugni benne a visususus

where the "template" specifies the syntax of the macro call and "body" is the token sequence that will replace the macro call at expansion time.

2.3 Macro Syntax: Part I

The set of macro calls that correspond to a given macro definition has a syntactic structure similar to regular expressions. Thus a template can be represented as:

template => [item] *

A template is a (possibly empty) sequence of items; each item represents either a parameter, a delimiter or a more structured construct called a regular expression.

When matching a template against the macro call, successive items are matched (according to the rules below). When the last item has been matched, the parsing of the macro call is considered successfully done.

If at any point an item cannot be matched against the currently scanned token, an error message is generated.

A delimiter is a keyword throughout a macro call; it delimits parameters and/or specifies the structure of regular expressions.

Parsing a delimiter item means matching the currently scanned token with the delimiter: they must be equal.

delimiter => token

Once the token is matched, the parsing process advances to the next item in the template and the following token in the currently scanned input stream.

parameter =>
 '<' parameter_name [balance] [environment] '>'

parameter name => token

A parameter is denoted by a name enclosed in angular brackets and represents a (possibly empty) sequence of tokens in the macro call. Parsing a parameter item implies successively scanning the input stream until a token is found that could match the next item in the template. In that respect parameters are "match-it-all" constructs: they match anything up to whatever the following parameter or delimiter can match (see Appendix A, Parsing Complexities).

Parameters can, however, be structured:

balance => [open_symbol close_symbol] +

open symbol => token

close symbol => token

The balance of a parameter is specified by pairs of tokens: every instance of "close_symbol" must be preceded by a matching occurrence of "open_symbol". This balancing process may apply to many pairs of symbols (with an error condition generated by sequences like "(a[b)]"...).

Whenever a parameter is being parsed, the currently scanned token is

- first checked to see whether the current parameter should be terminated;
- if not, the parser checks whether the token is a macro name, in which case it (possibly recursively) starts parsing the new macro call.

To determine whether the current token is a macro or not, the environment may be specified:

The environment tells the parser where to search for macro names:

- if the environment is empty, search in the active libraries (see Section 5.3, Library Handling)
- if the environment specifies a list of libraries, search those
- if the environment states "uses none" nothing within the parameter will be considered a macro

Example 1:

&let MAC &be ... &end sale dama sund medd

The above macro MAC defines only one macro call: the single token "MAC".

Example 2:

&let ALPHA BETA &be ... &end

The macro call for the macro ALPHA is the sequence "ALPHA BETA".

Example 3: lod do sarbag vd beittoega al nedemanag a do esas lad salt

&let do <QUESTION> . &be ... &end

Parameters match sequences of tokens; the following are valid macro calls associated with the macro "do":

do you wanna talk? OK. (<QUESTION> is
 associated with "you wanna talk? OK.")

do MAC . (<QUESTION> = MAC) bedshimmed ed bloods
do . (<QUESTION> is empty)

Example 4:

&let for <index> := <from> until <to>
 do <body do od> od
&be ... end

This macro defines an ALGOL-like for statement, that could be used to extend PLZ/SYS. Note that the parameter <body> has to be balanced with respect to pairs "do od", so that

"for i := 1 until 10 do vagas a management of do something od oddard as a management of do something oddard as a management of document of

will correctly match <body> to the sequence "do something od" (otherwise, the first "od" would match the delimiter "od" in the macro template, producing the unwanted parse).

Example 5:

&let if <exp uses expression_lib>
 then <true part> else <false part> fi
&be ... &end

This macro defines an "if" statement (as in PLZ). The first parameter, <exp>, has as its environment the library called "expression_lib" (presumably such a library will contain macro definitions that parse and generate code for the expression).

Example 6:

&let Simple_macro <P1> <P2> <P3>;

The above definition corresponds to an "old fashioned" macro with three parameters, and delimited by a semicolon. This example shows what happens with contiguous parameters: <Pl> matches the first token (or macro call), the following token is matched by <P2> and the rest, up to the semicolon, by <P3> (see Appendix A, Parsing Complexities).

2.4 Macro Syntax: Part II

The previous section dealt with the simpler kinds of items in a template. This section covers regular expressions.

regular_expression_name => token

separator => token

A regular expression is enclosed in square brackets followed by an optional count and consists of a set of choices. In order to identify the regular expression, a name is associated with it.

choice => [item]+

A choice is basically a non-empty template. Each choice within a regular expression represents an alternative form for the expression.

A regular expression also specifies a repetition: each instance of such repetition may take the form of any of the choices.

count => '+' | '*' | '?'

Both the separator and the count determine the number of occurrences of the regular expression in the actual macro call.

If the separator is non-null, the regular expression should occur at least one time; occurrences are separated by the separator. If the separator is null, the count determines what happens. The count may be one of the following, each with its corresponding meaning.

'+' : the regular expression occurs one or more times

'*' : the regular expression occurs zero or more times

'?' : the regular expression occurs zero or one times

empty : the regular expression occurs one time

The different types of regular expressions fall into one of two categories:

- those that may occur more than once (non-empty separator, or count being "*" or "+"); these are referred to as "repeat" expressions.
- those that may occur once at most (empty separator and count being empty or "?"); these are referred to as "choose" expressions.

Example 1:

&let Either [req: Yes | No | Maybe] &be ... &end

This is an example of a choose expression: the macro Either has three macro calls associated with it:

"Either Yes"
"Either No"
"Either Maybe"

Example 2:

&let call <proc> ([more ,: <arg>]) &be ... &end

Example 3:

&let Ml [loop: <Par>]*; &be ... &end

The macro Ml has a variable number of parameters (each one an instance of the parameter <Par>); parameters consist of single tokens. The repeat expression "loop" may have zero or more instances.

In the case of macro Read, the first choose expression matches an optional <format>, while the second is a repeat expression (one or more instances, separated by commas) of two choices: either the construct "array <arg>" or a parameter <scalar>.

those that may occur once at most (empty separator and count being empty or "2"); these are referred to as "choose" expressions.

Slet Either [reg: Yes | No | Maybe] She ... Send

This is an example of a choose expression: the macro Elther hree macro calls associated with it:

"Either Yes" "Either No" "Either Maybe"

The above macro matches procedure calls with one or more are are are are unents (separated by commas). The parameter 'proch will matche the procedure name while the repeat expression "more" will stand for the sequence of arguments, each one matched by the parameter

The macro Ml has a variable number of parameters (each one in instance of the parameter (ParX); parameters consist of single okens. The repeat expression "loop" may have seto or more

flet Read (<unit> (form:, <format>)?)
(read list ,: array <arg>

384

EXPANSION PROCESS

3.1 Body of a Macro: Part I

A macro body is a token sequence, with two special features that relate it to the corresponding macro call.

First, parameters are denoted by the construct "<name>". Whenever a parameter is reached at expansion time, the corresponding instance of that parameter in the macro call is expanded.

Example 1: disassumes disagraph for some of a some disagraph and

&let MAC &be MACKINTOSH &end | seems |

Every instance of "MAC" will be replaced by "MACKINTOSH".

But if we had defined some some soloval tola

&let MACKINTOSH &be Rain Coat &end

"MAC" would first be replaced by "MACKINTOSH" and "MACKINTOSH" would again expand into "Rain Coat".

Example 2: 14 gonoupes edit dadi sedada Peron dosesulu edi

This macro generates the PLZ/SYS code for a "for" loop. Note that no check is made as to the validity of the parameters.

3.2 Body of a Macro: Part II

In order to deal with regular expressions, the macro body must contain means to $% \left(1\right) =\left(1\right) +\left(1\right) +\left($

- access each instance of a repeat expression
- determine which choice has been made for each instance of a regular expression.

In order to access any instance of a repeat expression, the user must explicitly insert a "&foreach" construct in the macro body:

&foreach <repeat_name>
&do <foreach body> &od

where <repeat_name> is the name of a repeat expression and <foreach_body> is a token sequence that will be scanned once for each instance of the repeat expression.

Example: THOAM" vd base [der ad [[]]w "DAM" to especially where

CALL (proc) "1800 niss" ofni bnagwe nissa binew

&end

The "&foreach more" states that the sequence "PUSH <arg>" be scanned once for each instance of the expression "more". Furthermore, each time the &foreach is repeated, <arg> is associated with the corresponding parameter in the macro call. Thus

invoke PRINT_HEX (A, X+3, R4)

generates

PUSH A
PUSH X+3
PUSH R4
CALL PRINT HEX

386

In order to determine the choice selected for a given instance of a "choose" expression, the construct

<regular expression name>

evaluates to the leading token of the current instance.

Examples:

The macro

&let Either [req: Yes | No | Maybe] We got &if &eq str (<reg>, Yes) &then a positive answer &else &if &eq str (<req>, No) &then a negative answer &else no answer vet &fi s of copeat expressions, each i &fi

&end

will output three different sequences according to the choice made for the expression "reg". If <reg> is "Yes" the output will be "a positive answer"; if <reg> is "No", "a negative answer" will be output; if <reg> is "Maybe", "no answer yet" will be output. Note the use of the command "&if", which decides what to expand.

Similarly,

&end

&let Read (<unit> [form:, <format>]?) [read list ,: array <ary> - | <scalar> | *; &be Setup (<unit>) &if &isempty (<form>) &then Start_input (no_format)
&else Start_input (#<format>) &fi &foreach read list &do &if &eq str (<read list>, array) &then Array input (#<ary>) &else Scalar input (#<scalar>) &fi

In this case the construct <read_list> within the &foreach refers to the leading token of the current instance of the expression "read list", so that

Read (6, F str) A, array X, C[i];

produces the output

Setup (6)
Start_input (#F_str)
Scalar_input (#A)
Array_input (#X)
Scalar_input (#C[i])

3.3 Binding the Macro Call and the Macro Body

When expanding a macro whose template is composed only of parameters, delimiters, and choose expressions, a parameter or a choose expression can be referred to by naming it. However, in the case of repeat expressions, each instance of the expression has associated with it a different set of instances of the items defined within the expression. Thus, scope rules are needed to allow the expansion process to access different instances of items in the macro call.

"Accessing an item" within a macro body means:

- expand a parameter p (wherever the construct appears)
- evaluate the leading token of a choose expression C
 (by means of the construct <C>)
- execute a "foreach r" command for a repeat expression r and evaluate the leading delimiter of each repetition (with <r>).

The scope rules are:

- The items within a repeat expression r may be accessed only from within a "foreach r" command.
- 2. The items defined within a choose expression C (except for those internal to any inner repeat expression) may be accessed wherever C is accessible.
- 3. The items defined in the template of a macro (except for those internal to a repeat expression) are accessible from anywhere in the body of the macro.

The "foreach" construct ("&foreach r &do for_body &od") associated with a repeat expression "r" can be defined as follows:

- 1. The "foreach" may appear only where "r" is accessible
- 2. All the items accessible where the "foreach" is included are accessible from within "for body"
- 3. The "for_body" is scanned once for each repetition of the expression and all the items accessed are associated with the corresponding instances of themselves (in particular, the construct <r> within the "for_body" stands for the leading token of the current repetition).

ARITHMETIC: DATA STORAGE AND HANDLING

MacP provides the user with a facility to store arithmetic values and evaluate expressions. The command

&set <v> := <expression> &end

evaluates $\langle expression \rangle$ and associates the resulting value with a name denoted by $\langle v \rangle$. Further references to the token $\langle v \rangle$ within arithmetic expressions will yield its associated value.

MacP expressions are similar to PLZ/SYS expressions (for unsigned 16-bit WORD type). The operators handled by MacP are:

+, -, or, xor

*, /, mod, and

When MacP evaluates an expression, it first evaluates the "*", "/", "mod", and "and" operations, then applies the "+", "-", "or", and "xor" operations. This procedure parallels the standard PLZ precedence of operators. Precedence may be altered by means of parentheses.

Note that unary operators are not implemented.

The command

&out (<v>)

outputs the value of variable <v> as a five-digit decimal string.

COMMANDS

5.1 Conditional Expansion of Macros

The command

&if <exp> &then <textl> &else <text2> &fi

evaluates <exp> and scans <textl> or <text2> depending on whether its value is true or false.

The expression <exp> must be a call to either of the following three predicates:

> &isempty (<str>) true if <str> is the empty string; otherwise false

> &eq_str (<strl>,<str2>) true if <strl> and <str2> are equal; otherwise false

&isstring (<str>) false if (str) is a macro call; otherwise true

or a conditional expression of the form

<expl> <relop> <exp2>

where <expl> and <exp2> are arithmetic expressions and <relop> is one of declares the names given as libraries. After this command that saved, no more macro declarations of ic (nat case)

le (less or equal)

eq (equal)
ne (not equal)

gt (greater than) demands edl .allso closm was eximposes

ge (greater or equal)

Note: All macro commands in MacP are executed using the general Mote: All macro commands in the macro parsing mechanism.

5.2 Input Control

The command

&include <filename> &end

causes the scanning of <filename> to be started. When the endof-file condition is encountered, scanning of the previous file resumes. Nesting of &include commands is allowed up to five levels.

5.3 Library Handling (1989) ansoe bus cure asterious

A library is a collection of macros contained in a single file. Because of design considerations, MacP will only handle libraries that contain fewer than 255 macros.

Libraries are created by the following command:

&create <filename> &end

All the macro definitions found after a &create command will be included in the file specified. The previous content of the file is destroyed.

The current version of MacP is limited to 200 macros distributed in 32 libraries maximum.

The command

&libraries [C,: <lib>] &end

declares the names given as libraries. After this command is issued, no more macro declarations or &create commands are accepted by MacP.

Immediately after the &libraries command, MacP does not recognize any macro calls. The command

&activate [C, : <lib>] &end

causes MacP to recognize calls to all the macros included in the libraries specified (and only to these macros).

5.4 Miscellaneous Commands

The command

&unique

outputs a unique string of five decimal digits, from within a macro expansion. The particular string remains constant throughout the entire expansion. For instance:

&end

A call like goroom to moit/mided

do
inc hl
ld a (hl)
and a
ld (hl) a

od

will generate

L01937:

inc hl

id a (hl)

and b

id (hl) a seed system of the property of th

where the string "01937" was arbitrarily chosen and could be any string of five decimal digits.

The command

&peel [c: on|off] &end and production of the second

instructs the scanner to either discard double quotes enclosing strings (&peel on &end) or to leave quotes (&peel off &end).

By default, during the first phase (library creation), peeling is set to "off" (so double quotes remain inside of macro bodies). During expansion phase, the peeling is set to "on" (so double quotes are not output).

SECTION 6

MacP OPERATION

MacP consists of two phases. The first phase compiles macros and creates libraries. The second phase uses existing libraries to expand the input stream.

Both phases or only one phase may be used in any single run of MacP; if both, the macro generation phase must be the first one.

An example of the input stream is as follows:

&create libl &end

&let macll ... Definition of macros &let macl2 ... included in libl

&create lib2 &end

&let mac21 ... Definition of macros &let mac22 ... included in lib2

&create lastlib &end

Definitions of macros for lastlib

&libraries libl, lib2, ... &end &expand

!No macros are active here, so no expansion will occur!

&activate libl, lib2, ... &end

Text to be expanded using the macros in libraries libl, lib2,

Note that the command "&libraries" starts the expansion phase. Also, if existing libraries are used, the library creation phase does not need to be initialized.

MacP accepts input files with the suffix .MAC or .mac. To invoke MacP, type to RIO the following command:

MACP file tames serioup elduob oza "alto" oz des el parteso

When processing a file X.MAC, MacP will produce two files:

- An input image file, X.M, including line numbers and error messages
- The output file, X.S

Similarly, X.mac will produce X.m and X.s, respectively.

APPENDIX A

PARSING COMPLEXITIES

This section introduces some of the ambiguous situations that may arise during the parsing of macro calls, and presents the conventions adopted to solve them.

0 - Nested Macro Calls: during the parsing of a parameter, a macro name is encountered.

Rule 0: Proceed to parse the entire macro call (recursively) and match the entire macro call to the parameter.

Example:

Given the macro

&let if <exp> then <tpart>
 [:else <fpart>]? fi &be ... &end

A macro call such as

if x < 0 then x := 0
 else if x = 0 then exit fi
 call interp fi</pre>

matches

1 - Delimiter vs. Macro Name: during the parsing
 of a parameter, a token is encountered that is both
 a delimiter and a macro name.

Example:

&let Loop <index> begin <body> end

&be ... &end &be ... &end

Rule 1: The token is considered a delimiter if it terminates the parameter; otherwise, it is considered a macro name.

In the following case:

Loop i begin A[i] := 0 end

both "begin" and "end" are treated as delimiters.

But in the case

Loop i end begin A[i] := 0 end

the first "end" will be treated as a macro call and the second one as a delimiter.

2 - Parameter after Parameter: two parameters are contiguous in a template. How to match successive tokens in the two parameters is explained below.

Example:

&let M <pl> <p2> ; &be ... &end

Rule 2: The first parameter matches one token (if any) and then the parse continues with the contiguous parameter and the rest of the tokens.

In the following cases:

- (1) M al a2 a3 a4;
- (2) M bl:
- or (3) M:
- in (1) $\langle p1 \rangle = a1$, $\langle p2 \rangle = a2$ a3 a4
- in (2) $\langle p1 \rangle = b1$, $\langle p2 \rangle$ is empty in (3) both $\langle p1 \rangle$ and $\langle p2 \rangle$ are empty

Note that Rule 2 also applies to

&let Ml [seq :]* ; &be ... &end

so that

Ml abc:

matches a, b and c to three instances of .

Also, the case

M Ml x y; toktok;

will match

 $\langle pl \rangle = Ml \times y ;$ and $\langle p2 \rangle = toktok$

since Ml is a macro name.

- 3 Macros that cannot be parsed: Given the parsing method used by MacP, some macro definitions cannot be parsed without resorting to a guess somewhere along the parse. MacP rejects these macros when processing their definition and warns the user.
- Rule 3: MacP accepts only macro templates that can be parsed by looking at most one token ahead.

Examples:

&let Mac [ch : a x | a y] &be ... &end

When parsing the expression "ch", MacP must look $\underline{\mathsf{two}}$ tokens ahead ("a" and the next one) in order to determine which choice to take.

&let Mic [loop :]* <pl> ; &be ... &end

The libraries declared in 'Slibraries' contain too many macros
Problems in the library [1]e (macro.

In the above example, there is no way to decide whether the next token is a new instance of the repeat expression "loop", or is a match for the parameter "pl".

APPENDIX B

ERRORS

MacP produces errors of different types:

| W | (Warning) | An error is detected from which MacP can recover. The user output may not be correct but execution continues. |
|---|-----------|---|
| S | (Serious) | Recovery may or may not be possible. |
| F | (Fatal) | No recovery is possible, MacP aborts. |

Additionally, MacP will enter a loop at a certain address when an undefined error occurs. If your system enters a loop, most likely you have found a bug in MacP.

The following is a list of the error codes produced by MacP and their type.

| Error | Type | Condition |
|-------|------|---|
| 20 | (F) | Buffer overflow (either for macro generation or for library's dictionary) |
| 21 | (F) | Too many libraries (more than 32) |
| 22 | (F) | The libraries declared in '&libraries' contain too many macros |
| 23 | (F) | Problems in the library file (macro header is wrong) |
| 24 | (F) | Macro name too long (more than 32 bytes) |
| .25 | (F) | Bad input file name (must end with '.mac' or '.MAC') |
| 26 | (F) | '&libraries' command issued from within a macro |
| 27 | (F) | Too many nested '&includes' |
| 28 | (F) | Same as Error 21 |
| 29 | (F) | Duplicate library name in the '&libraries' command |
| 30 | (F) | Not enough buffer space to load a macro |
| 31 | (F) | Too many active macros in memory (more than 100) |
| 33 | (S) | A library that appears in the 'uses' part of a parameter or the list of an '&activate' command has not been declared in the '&libraries' list |
| 35 | (S) | Too many libraries in the list of '&activate' |

| Error | Type | Condition ones agy E agy E |
|----------------|-------------------|---|
| 50 51 52 | (W) (S) (S) | Missing parameter name Non-empty count required The name between angle brackets is not a parameter or a regular expression name |
| 53 | (S) | |
| 54 | (S) | Too many delimiters in the macro |
| 57 | (S) | The name has already been declared as a |
| 58 | (S) | parameter or a regular expression name Too many parameters and/or regular expressions |
| 61 | (S) | No closing balance delimiter |
| 62 | (S) | Too many libraries used in this macro |
| 63 | (F) | Macro generation buffer overflow |
| 64 | (F) | Too many items in current macro template |
| 65 | (S) | Name is not a repeat expression: &foreach is ignored |
| 66 | (S) | Repeat expression used in enclosing |
| 67 | (S) | &foreach: &foreach is ignored. Unknown external procedure in '&call' |
| 74 | (W) | Ambiguous syntax: regular expression may be empty |
| 75 | (W) | Syntax problems in separator list: conflict with next item's lead |
| 76 | (F) | Too many delimiters |
| 77 78 | (S) (W) | Balance problems Syntax problems in choice list: conflicting leads |
| 90 | (F) | Parsing nesting too deep |
| 91 | (F) | Parse stack overflow |
| 92 | (S) | Balance pair does not match |
| 93 | (S) | Syntax error (during macro call parse) |
| 100 | (F) | Input buffer overflow |
| 101 | (F) | Input error |
| 102 | (F) | Too many nested macro |
| | | expansions/&includes |
| 103 | (F) | Too many '&includes' |
| 104 | (S) | I/O error when closing include file |
| 120 | (F) | Evaluation stack overflow |
| 121 | (F) | Boolean expression expected |
| 122 | (F) | Stored values name pool overflow |
| 123 | (F) | Operand name cannot be evaluated |
| 124 | (F) | Too many stored values |
| 125 | (S) | Undefined value (evaluates to zero) |

| Error | Type | Condition 5000 |
|---------------------------------|------------------------------------|---|
| 126 | er name (F) | Expression does not yield an arithmetic value |
| 127 200 201 | (F) and alpha d (W) a salapas a | Expression is ill defined Decimal number overflow |
| 202 | (W) | Invalid hexadecimal digit |
| 204 | (S) meed vosen | Unmatched quote |
| 200 201 202 203 204 | (F) (W) (W) (W) (S) | Expression is ill defined Decimal number overflow Hexadecimal number overflow Invalid hexadecimal digit Invalid character |

APPENDIX C

I/O UNITS

The following table lists the MacP Input/Output units and their usage.

| Unit mass and an heiti | Usage |
|------------------------|--|
| 4 | "&create" library output |
| 5 | Listing |
| 6 | Expansion output |
| 7 | Macro loading (input) unit |
| 10-14 | Input files (initial one is 10, and each nested &include is assigned successive units. When a file ends its unit is freed) |

APPENDIX D

OUTPUT FORMAT

In MacP, a file with no macro calls produces an output of the same format. But in the case of a new macro expansion the output will not reflect the spacing between tokens. Therefore, the spacing must be explicitly specified in the macro definition. For instance, the macro

&let M <pl> , <p2> ; &be <pl> <p2> &end

and a call like

linel M John, Alfred; back could be line2

will produce the output

linelJohnAlfredline2

A more desirable output format might be

linel

John Alfred

line 2

which is achieved using sequences of spaces, tabs and carriage returns enclosed in double quotes (").

&end

Note: The string after &be is "<CR><TAB>"

Z80/Z8000 Translator User Guide

Decomposition Pass

CONTENTS

| SECTION | 1 | INTRODUCTION 40 | | |
|----------|---|--|--|--|
| SECTION | 2 | TRANSLATOR STRUCTURE 408 | | |
| | | 2.1 2.2 2.3 2.4 | Decomposition Pass | 408 408 409 410 |
| SECTION | 3 | TRANS | 58000 OPERATION | 412 |
| | | 3.1 3.2 3.3 3.4 3.5 3.6 | Mappings User Controls I/O Handling Errors | 412 413 414 414 415 415 |
| | | | 3.6.2 Step | 415 416 417 418 |
| | | 3.7 | Implementation Note | 419 |
| SECTION | 4 | TABL | ES | 4:20 |
| | | 4.1 4.2 4.3 | | 420 426 427 |
| APPENDIX | Α | EXA | MPLES | 431 |
| | | A.1 A.2 | General Example | |

SECTION 1

INTRODUCTION

The Z-80/Z8000 translator is a programming tool for upgrading RIO Z-80 programs to the Z8000 instruction set and the PLZ/ASM program format.

 $\tt TRANS8000$ is designed to run under the RIO operating system on both the MCZ and ZDS systems. A minimum memory size of 48K is required for TRANS8000 to execute.

The following manuals may be useful to the reader:

- Z-80 RIO Relocating Assembler and Linker User's Manual
- Z-80 Assembly Language Programming Manual
- Z-80 RIO Operating System User's Manual
- Z8000 PLZ/ASM Assembly Language Programming Manual

SECTION 2

TRANSLATOR STRUCTURE

2.1 Overview management and all modeledent

The translation process makes two distinct passes. The first pass, called the Decomposition pass, converts each line of Z-80 ASM source to an intermediate form and symbols are entered into the symbol table with scope and type information. The second pass, called the Building pass, builds the Z8000 PLZ/ASM source and critique files.

The translator may be executed in either an Interactive mode or in a Noninteractive mode. This allows the programmer to set the translation options along with flag and register mappings at translation time.

2.2 Decomposition Pass

During the decomposition process, each field of a source line is scanned and denoted as a label, a pseudo-op, an opcode or an operand. Operands can be either keywords or expressions. The Z-80 source lines are not checked for legal assembly language syntax nor for correct opcode and operand combinations. Decomposition processing of a line continues until a carriage return or a comment field is recognized.

Many of the Z-80 opcodes are dependent upon the operand combination. These opcodes are fixed up after the source line has been processed because the equivalent Z8000 instruction does not allow the same set of operands (usually indirect register addressing), or the opcode does not uniquely specify whether the operation handles 8 or 16-bit quantities.

Pseudo opcodes also need fixing to reflect their meaning. The "LIST" pseudo-op will be changed according to its operand. The "ORG" pseudo-op will be changed to either "\$ABS" or "\$REL". Keywords in the decomposition process are matched with a case of all upper or all lower depending on the case of the first letter of the keyword.

Macros are not processed during the Decomposition pass. Likewise, no reconstruction processing of macros is done during the building process.

Symbols in expressions are translated with the rest of the syntax. Most valid Z-80 operators are also valid in Z8000 PLZ/ASM expressions. Any occurrences of invalid operators cause a warning message to be displayed. For example, the two Z-80 arithmetic operators "**", for exponentiation, and ".RES.", for overflow conditions, do not have PLZ/ASM equivalents. Also, when using the program counter variable, "\$", in expressions, the user should consider that Z-80 instruction sizes are not the same for equivalent Z8000 instructions, and that the precedence order of logical and arithmetic operators is not the same.

Because Z8000 PLZ/ASM handles all expressions as 32-bit unsigned quantities, signed greater-than and signed less-than Z-80 ASM operators are translated to "reasonable" PLZ/ASM expressions and noted in the critique file.

2.3 Building Pass

The translator performs two distinct functions. One is to take a sequence of Z-80 instructions and change them to a sequence of Z8000 instructions while preserving the operational equivalence of the execution. The other is to convert the syntax of the Z-80 assembly language, including opcodes, operands, pseudo-ops, and expressions, to the syntax of the Z8000 PLZ/ASM assembly language. The first function will not be dealt with completely. Approximately 96% of the Z-80 instructions can be translated to operationally equivalent Z8000 instructions. Most of the translations are one for one (about 80%), the rest are one to many. Operational equivalence means that the register values and flag states are left in equivalent states after the operation. Operational equivalence does not imply that the number of bytes and the CPU time necessary to simulate the operation are equal.

The function of converting syntax is dealt with completely during the building pass. All opcodes, operands, pseudo-ops, and expressions are changed to Z8000 PLZ/ASM syntax. Also, the line-oriented Z-80 assembly language format is fitted into the structured assembly language as defined by PLZ/ASM.

Since PLZ/ASM is a structured assembly language, TRANS8000 must put the Z-80 source program into a structured form. The objective is to leave instructions and declarations in the original order and insert structuring information where appropriate. The entire program is then formed into a module. The module name is prefixed by the source file name with an "_MOD" appended. Definitions are handled by appropriate declaration sections. All contiguous instruction lines (including blank, comment and assembler control lines) are grouped into one internal procedure. A generated procedure name, "PROC_nn", is assigned to satisfy the PLZ/ASM syntax. Any labels

appearing within that procedure are implicitly declared as internal unless declared elsewhere. Labels declared as global or external are explicitly declared as such at the original point in the program. A sequence of instructions broken by a data declaration cause the ending of the previous procedure and the starting of a declaration section. Any subsequent instructions cause a new procedure to be entered and a new procedure name to be generated.

Most Z-80 instructions and Z-80 ASM pseudo-ops will be translated to an equivalent form. The translation is done line-by-line to preserve ordering of code and declarations. There are many cases where one Z-80 instruction will generate several Z8000 instructions, particularly if the user requires that the flag states remain equivalent for each line of the execution. Flag incompatibility and fixups are discussed in detail in Section 4.3. Instructions that are not translatable are those most directly related to the differences in CPU architecture. Because PLZ/ASM does not support macros, all macro definitions are passed through the translator unprocessed. An approach to solving this problem is discussed below.

There are two problem situations that may arise for which the translator has no reasonable solution and must therefore be corrected by the user. One problem involves a small group of instructions that have no logical equivalant due to architectural differences between the Z-80 and Z8000 Central Processing Units. The second problem involves an instruction sequence that accesses a memory byte logically declared as a portion of a word quantity. This problem occurs because the Z-80 stores word quantities in memory with the high and low bytes in reverse of the Z8000 memory layout.

Frequently in programming with the RIO Z-80 assembler, a programmer would equate a label to an address constant. The label would then be used in a Call or Jump command in place of the constant. The PLZ/ASM assembler imposes stricter restrictions upon the use of a constant. Only constants declared to be labels may be used in a Call or Jump command. The translator will not correct this problem for the user, but the user may easily fix this by editing the Z8000 source file.

2.4 Handling Macros

An important difference between the Z-80 ASM and Z8000 PLZ/ASM is that the latter does not support a macro capability.

The problem of translating a program that is heavily dependent on the macro facility available in the Z-80 assembler can be handled as follows: Assemble the program creating a

listing file with the macros expanded, then remove the unwanted information (opcodes, object bytes, etc.) using the Editor. This new file may then be processed by the translator.

SECTION 3

TRANS8000 OPERATION

3.1 Command Line Format

The format for the translator command line is as follows:

TRANS8000 source-filename [options]

The first item in the command line after the program name, "TRANS8000", must be the filename of the file to be translated. The file must have a file extension of ".S" or ".s". The given filename may be a fully or partially qualified RIO filename.

All files created by TRANS8000 will consist of the given filename, with the ".S" removed, and the appropriate extension appended. The following extensions are used for the respective files:

.Z8000.S => translated Z8000 PLZ/ASM source

.CRIT => critique file

.INT => intermediate file (temporarily created)

Any of the above filename defaults can be overridden by the command line options "S=", "C=", or "I=". Directing file output to device \$NULL causes no file to be written.

The command line can also contain any of the directives listed in Section 3.3, except the "STEP", "FLAG", and "REGISTER" directives.

EXAMPLE:

%TRANS8000 DEMO DISPLAY C=\$NULL

causes the file DEMO.S to be translated with the source displayed at the console and no critique file written.

When the translator is invoked from RIO, an echo of the TRANS8000 name and the version will be displayed at the console. The version echo tells the user which release is being executed, and when the execution has begun. Also, there is an echo that informs the user of the completion of the decomposition and building passes. Upon completion of the translation process, all files are closed and the message "TRANSLATION COMPLETE" is displayed at the console.

3.2 Mappings

There are two user-definable mappings to associate Z-80 and Z8000 CPU resources: the flag and register mappings.

The flag map is a means of altering the generated Z8000 source code. For each Z-80 instruction that is translated, extra Z8000 instructions are added to keep the CPU flags compatible. The user can state which flags are to have extra code generated for them ("care" state) and which flags are not to generate extra code ("don't care" state). The user should determine whether the program to be translated has been coded to rely on particular CPU flags and set the mappings accordingly. Interactive and source directive capabilities provide control of isolated sections of flag critical code. The default flag mappings are:

```
C Care
Z Care
S Care
P/V Don't care
N Don't care (called "D" flag in the Z8000)
H Don't care
```

These default mappings can be altered by the "FLAG" directive.

The Z-80 CPU registers must be associated with Z8000 CPU registers. Sixteen and eight-bit registers must be mapped appropriately to allow compatible execution. TRANS8000 does not check to insure that a logical association of registers is maintained. The default register mappings are:

```
RHO F - RIO AF - RO
  - RH1 C - RL1 BC - R1
B
                   DE - R2
        E - RL2
D
    RH2
                    HL - R3
SP - R15
            - RL3
          L
H
  - RH3
          IY - R14
IX - R13
                    DE' - R10 pail bash OVI
          BC' -
              R9
AF' - R8
HL' -
    Rll
```

These default mappings can be altered by the "REGISTER" directive.

A list of free registers is acquired at the beginning of each pass and whenever a register map is altered. When a scratch register is needed to build Z8000 source instructions, one of the free registers can be used. The scratch register list is gathered by scanning the register map, starting with R0, and keeping any register not previously mapped. R7 is the highest numbered register in the scratch list because 8-bit registers are used for the scratch registers. The scratch registers are used only in the translation of Z-80 CPU instructions. They can be re-used for each instruction.

3.3 User Controls

The user is given control over the translator by the use of the options and directives. The areas of user control over the translator are:

- 1. Generate relocatable or absolute Z8000 PLZ/ASM source
- 2. Display and set the register map
- 3. Display and set the flag map
- 4. Display Z-80 source, Z8000 source, and messages on the console
- 5. Include the Z-80 ASM source in the critique file
- 6. Execute TRANS8000 in Interactive or Noninteractive mode
- 7. Translate interactively on Pass One or Pass Two
- 8. Step through the translation by lines
 - 9. Halt translation and return to the system

The defaults are provided to generate relocatable code, to include the Z-80 ASM source in the critique file, to display only messages on console, and to execute in the Noninteractive mode.

All file and console output interfaces with RIO through STREAM (which is a part of the TRANS8000 program). Console input allows character deletion: <backspace> key (ASCII %08), and line deletion: <delete> key (ASCII %10).

The logical unit assignments made by TRANS8000 are:

Unit 2 Console

Unit 4 Intermediate file

Unit 5 Z8000 source file

Unit 6 Critique file

Unit 7 Z-80 source file

All messages from the translator will appear at the console and in the critique file. Messages pertaining to a given source line will appear after that line has been processed.

3.5 Errors

When TRANS8000 determines that an execution error has occurred, processing will stop. An execution error is not directly related to the content of the source file, but is related to the environment in which the translator is running (memory availability, disk storage, etc.). When a condition arises that causes the translator to abort, the following events occur:

- 1. any buffered messages get displayed,
- 2. a hexadecimal abort-code is displayed,
- 3. all files are closed,
- 3. all files are closed,
 4. execution returns to the system.

The abort-code can be either a RIO completion code or a TRANS8000 abort number (see Section 3.7, Implementation Note for code descriptions).

The translation process can be aborted at any time by hitting the Escape (esc) key. 3.6 Directives

Options are set using directives. Directives can be entered via the command line, the Z-80 ASM source file (if the line is preceded by a "%" character), or via the console, in the Interactive mode.

3.6.1 Description and made and a second seco

TRANS8000 prompts the user with a "T>" when console input is ' expected. Console input is always terminated by a carriage return. The directives that control TRANS8000 are:

DIRECTIVE

"ABS" "BRIEF" "DISPLAY"

"FLAG" "INTERACT" "TP]"

"TP2"

"REGISTER"

"VERBOSE"

Generate absolute Z8000 source

Exclude Z8000 source in critique file Display Z-80 and Z8000 sources at the

console

Display or set flags Enter Interactive mode

Execute interactively during Pass One only Execute interactively during Pass Two only

"NODISPLAY" Inhibit sources from appearing at the

console

"NOINTERACTION" Leave Interactive mode
"OUIT" Halt execution of TRANS8000

Display or set register mappings "RELOC" Generate relocatable Z8000 source
"STEP" Translate lines interactively

Include Z8000 source in critique file

Directives can be entered as all uppercase, all lowercase, or mixed upper and lower case. All directives can be abbreviated to their shortest unique representation. More than one directive, separated by valid delimiters, may be entered on a line. If the same directive is entered more than once on a line, only the rightmost occurrence executes. Note that in the following descriptions, the user should type the symbols enclosed in double quotes.

The "FLAG" and "REGISTER" directives both have qualifiers for setting flag and register maps, respectively.

3.6.2 Step de Bedrode ad man aasterd mold

The "STEP" directive allows the user to alter the flag or register mappings and other translator states at any point during the process. The "STEP" directive accepts three qualifiers: a number, a label, or an asterisk. A number qualifier, ranging from zero to 255, causes repeated processing of source lines until the count is exhausted. A label qualifier, six characters or less, will step until the line containing that label at its start has been processed. An asterisk qualifier will process until the end of the current pass. If no qualifier is specified, a number qualifier of one is implied. A directive line appearing in the source counts as a line when stepping. Also, "STEP" commands will not be processed if a Step command is already active. That is, no recursion of the Step command is allowed.

A number qualifier may be entered with radices of two, eight, ten, or sixteen by appending the appropriate base indicator after the number. No appendage implies a decimal value; the other appendages explicitly imply:

```
"b" "B" - binary
"o" "O" - octal
"q" "Q" - octal
"d" "D" - decimal
"h" "H" - hexadecimal
```

EXAMPLE:

```
T>STEP 5 (steps 5 lines)
T>STEP 0CH (steps 12 lines)
T>STEP * (steps to end of current pass)
```

3.6.3 Flag

The flag directive can be used to display the flag state or to set the flag states. The directive keyword "FLAG" by itself will display the current flag state. The directive keyword "FLAG" followed by a flag setting list enclosed in parentheses will set the flag states. The form of the flag directive is:

If any flag-name or flag-setting is duplicated, the rightmost occurrence will take precedence.

The default setting for a list without a flag-setting specified is "ON". The "ON" keyword means that the flag is in a Care state, while the "OFF" keyword means the flag is in a Don't Care state.

EXAMPLE:

T>FLAG (Z C ON S OFF)

This flag directive sets the Zero and Carry flags to Care, sets the sign flag to Don't Care, and does not affect the Parity, Half Carry, and Decimal Adjust flags.

3.6.4 Register

The register directive can be used to display the register mapping or to set the register mappings. The directive keyword "REGISTER" by itself will display the current register mapping. The directive keyword "REGISTER" followed by a map setting list enclosed in parentheses will set the register map. The form of the register directive is:

```
register-directive => "REGISTER"
                      => "REGISTER" "(" list * ")"
                list => Z-80-reg-name "=" Z8000-reg-name
     Z-80-reg-name => "B" | "C" | "BC" | "BC""
                      => "B" | "C" | BC | BC
=> "D" | "E" | "DE" | "DE!"
=> "H" | "L" | "HL" | "HL'"
=> "A" | "F" | "AF" | "AF'"
=> "IX" | "IY" | "SP"
    Z8000-reg-name => RH0 | RL0 | R0
                      => RH1 | RL1 | R1
                      => RH2 | RL2 | R2
                      => RH3 | RL3 | R3
                      => RH4 | RL4 | R4
                      => RH5 | RL5 | R5
                      => RH6 | RL6 | R6
                      => RH7 | RL7 | R7
                              R9 R10 tage paid and the flag
                      => R8
                      => R11 | R12 | R13
                      => R14 | R15
```

any register-mapping is duplicated, the rightmost occurrence takes precedence. Also, only mappings specified alter the current mappings.

All 8 and 16-bit registers must be mapped individually. Therefore, mapping a 16-bit register pair does not imply that the two corresponding 8-bit registers, if they exist, are also mapped. Cara state ship of the "OFF" keyword means the flag is all MAXE

T>REG (A=RH4 F=RL4 AF=R4)

maps the two 8-bit registers and the 16-bit register of the Z-80 and Z8000 together.

3.7 Implementation Note

The abort codes displayed by TRANS8000 are listed below. Codes in the range %20 to %3F are generated by TRANS8000. Codes greater than %3F are ZDOS completion codes. Contact Zilog if any of the following codes are returned.

| CODE | MEANING |
|--|--|
| 20 21 22 23 24 25 26 27 28 29 2A 2B 2C 2D 2E | Intermediate file out of sync or invalid record header in intermediate file Source directive sync error Intermediate record header error See 21 See 22 See 21 See 22 Scratch list overflow Request for scratch register overflow No scratch registers available Scratch register overflow Symbol table overflow Invalid next state while restructuring Source item buffer overflow |
| 2F 30 31 32 33 | Data name generator overflow Procedure name generator overflow Zero flag fixup error Sign flag fixup error Parity/overflow fixup error |

SECTION 4

TABLES 9304 nordspreading v

4.1 Opcode Tables

The opcode table below lists the Z-80 instructions (with some repetition omitted) and the equivalent instruction(s) produced by the translator.

Instructions marked "NOT TRANSLATABLE" result from lack of architectural compatability. Registers are dependent on register mapping. Instructions using IY are not shown (because they are similar to IX).

NOTATION:

| r,r' | Any Z-80 registers |
|-----------|--|
| rb,rb' | Any Z8000 8-bit register (high or low), $b = 0$ to 7 |
| rn aldali | Any Z8000 16-bit register, $n = 0$ to 15 |
| n | An 8-bit value |
| nn | A 16-bit value |
| d wolls | A displacement value |
| e | Signed two's complement number in range <-128,127> |
| SS | Valid 16-bit operands for the given Z-80 instruction |
| m | Valid 8-bit operands for the given Z-80 instruction |
| VV | Valid operands for given Z8000 instruction, explicitly shown for preceding instruction |

OPCODE TABLE

| Z-80 Instruction Equi | valent Z8000 Instruction |
|---------------------------|--------------------------|
| LD r,r' | LDB rb,rb' |
| LD r,n | LDB rb, #n |
| LD r, (HL) | LDB rb,@rn |
| LD r, (IX+d) | LDB rb,d(rn) |
| LD (HL), r | LDB @rn,rb |
| LD (IX+d),r | LDB d(rn),rb |
| LD A, (BC) | LDB rb,@rn |
| LD A, (DE) | LDB rb, @rn |
| LD A, (nn) | LDB rb,nn |
| LD (BC),A | LDB @rn,rb |
| LD (DE),A | LDB @rn,rb |
| LD (nn),A | LDB nn,rb |
| | *NOT TRANSLATABLE |
| LD A, R | *NOT TRANSLATABLE |
| LD I,A | *NOT TRANSLATABLE |
| XOR m | XORB rb, vv |
| CP m | CPB rb, vv |
| INC r | INCB rb,#1 |
| INC (HL) | INCB @rn,#1 |
| INC (IX+d) | INCB d(rn),#1 |
| DEC m | DECB vv, #1 |
| ADD HL,ss | ADD rn,rn |
| ADC HL,ss | ADC rn,rn |
| SBC HL,ss | SBC rn,rn |
| ADD IX,ss | ADD rn,rn |
| INC ss | INC rn,#1 |
| INC IX | INC rn,#1 |
| DEC ss | DEC rn,#1 XI, MARKE |
| DEC IX | DEC rn,#1 |
| RLCA HINDID HILL | RLB rb,#1 |
| RLA TIME STATE STATE | RLCB rb,#1 |
| RRCA HAMIL HAM | RRB rb,#1 |
| RRA Dalamana, da algi | RRCB rb,#1 |
| RLC r | RLB rb,#1 |
| | |
| RLC (HL) | LDB rb,@rn |
| | RLB rb,#1 |
| | LDB @rn,rb |
| | |
| RLC (IX+d) [MI] Bydl Eddy | |
| | |
| | LDB d(rn),rb |

| Z-80 Instruction | Equivalent Z8000 Instruction |
|------------------|------------------------------|
| RL m | RLCB VV |
| RRC m | RRB VV |
| RR m | RRCB VV |
| | SLAB VV |
| D LIT III | SRAB VV |
| SRL m | SRLB VV |
| | 1) 6 EQU 1. (6+XI) QU |
| | da add LDB rb,@rn |
| | da RLDB rb, rb |
| | da adaLDB @rn,rb (ma) A da |
| | |
| LD R,A | *NOT TRANSLATABLE |
| LD ss,nn | and Ed LD rn, #nn |
| LD IX, nn | III TOW LD rn, inn |
| LD ss,(nn) | AT TOW LD rn, nn |
| LD (nn),ss | AT TOW LD nn, rn |
| LD SP, HL | da agoxLD rn,rn m go |
| LD SP, IX | da ago LD rn, rn |
| I ODII dd | PUSH @rn,rn |
| | as admiror rn,@rn |
| EX DE, HL | B SOMIEX rn, rn B+XI) DM |
| EX AF, AF | WV BOBOEX rn,rn |
| EXX | And ddA |
| 222 | EX rn,rn |
| | EX rn,rn |
| | |
| EX (SP), HL | EX rn,@rn |
| EX (SP), IX | na oad EX rn,@rn |
| LDI | LDIB @rn,@rn,rn |
| LDIR | LDIRB @rn,@rn,rn |
| LDD | da aogg LDDB @rn,@rn,rn |
| LDDR | da 899 LDDRB @rn,@rn,rn |
| | da HOMA CPIB rb,@rn,rn,EQ |
| CPIR | da Ala CPIRB rb,@rn,rn,EQ |
| CPD | CPDB rb,@rn,rn,EQ |
| | da add CPDRB rb,@rn,rn,EQ |
| | da Ban ADDB rb,rb |
| ADD A, n | |
| ADD A, (HL) | ADDB rb,@rn |
| ADD A, (IX+d) | |
| | ADCB rb,vv |
| ADC A, (HL) | ADC rb,rb |
| ADC A ~ | LDB rb,#n |
| ADC A,n | ADC rb,rb |
| | ADC ID, ID |

| Z-80 Instruction | Equivalent | Z8000 | Instruction |
|-------------------|------------|-------|-------------|
| E OU THECT GOCTON | 2442142 | 2000 | |

| OO INDELACEION | |
|--|--|
| ADC A,n | LDB rb,@rn ADC rb,rb LDB rb,#n ADC rb,rb |
| AND m OR m | ANDB rb,vv |
| RRD | 122 C211/22 |
| CPL NEG CCF SCF NOP HALT DI EI IMO IM1 IM2 BIT b,r BIT b,(HL) BIT b,(IX+d) SET b,r SET b,(HL) SET b,(IX+d) RES b,m JP nn JP CC,nn JR e JR C,e JR NC,e JR Z,e | DAB rb COMB rb NEGB rb NEGB rb COMFLG C SETFLG C NOP HALT DI VI EI VI *NOT TRANSLATABLE *NOT TRANSLATABLE *NOT TRANSLATABLE *NOT TRANSLATABLE BITB rb,#b BITB d(rn),#b BITB d(rn),#b SETB rb,#b SETB rb,#b SETB d(rn),#b |
| (, | HAMID JP @rn HAMMI JP @rn |

| Z-80 Instruc | tion Equiv | valent Z8000 Instr | uction |
|-----------------------|------------|--|--------|
| DJNZ e | | DBJNZ rb,e backwards jump on forward jumps not | |
| CALL nn | | | |
| CALL CC,n | | | |
| RET RET CC RETI | | RET CC | |
| RETN RST n | | *NOT TRANSLATABLE *NOT TRANSLATABLE | |
| IN A,(n) | | INB rb,n | |
| IN r,(C) | | CLR rn LDB rb,rb INB rb @rn | |
| | | | |
| INIR | | CLR rn LDB rb,rb CLRB rb INIRB @rn @rn rn LDB rb,rb | |
| IND | | CLR rn LDB rb,rb CLRB rb INDB @rn @rn rn LDB rb,rb | |

INDR CLR rn
LDB rb,rb

CLRB rb
INDRB @rn @rn rn
LDB rb,rb
CLRB rb
INDRB @rn @rn rn
LDB rb,rb

OUT (n),A

OUTB n,rb

LDB rb,rb OUTB rb @rn

Z-80 Instruction Equivalent Z8000 Instruction

OUTI CLR rn LDB rb, rb CLRB rb

OUTIB @rn @rn rn

LDB rb, rb

OTIR

CLR rn
LDB rb,rb
CLRB rb
OTIRB @rn @rn rn
LDB rb,rb

OUTD

CLR rn LDB rb,rb CLRB rb CLRB rb
OUTDB @rn @rn rn

LDB rb, rb

OTDR

CLR rn LDB rb,rb CLRB rb

OTDRB @rn @rn rn

Manga a g LDB rb,rb

4.2 Pseudo-Ops and UDDBM the sweeps and doubt alend Ud-S

The Z-80 pseudo-ops and their equivalent Z8000 syntax are listed below.

| Z-80 Pse | udo-Ops | nr9 811 | Equivalent Z8000 Syntax |
|---|--------------------|---------|--|
| | n n 'string' | | V1 BYTE := n V2 WORD := n V3 ARRAY [n BYTE] V4 ARRAY [* BYTE] |
| V5 DEFT | 'string' | | := 'string' V5 ARRAY [* BYTE] := '%nnstring' |
| V6 EQU V7 DEFL GLOBAL | n na nag | | CONSTANT V6 := n No translation GLOBAL |
| EXTERNAL ORG | | | \$ABS or \$REL |
| COND n ENDC | | | \$IF n \$THEN \$FI |
| MACRO ENDM | | | No translation |
| *EJECT *HEADING *LIST ON *LIST OFF *INCLUDE *MACLIST *MACLIST | file ON | | \$PAGE \$TITLE string \$LISTON 1 \$LISTON 0 \$INCLUDE 'file' No translation No translation |
| +, -, *, | /, MOD | | +, -, *, /, MOD |
| .NOT., .F | AND., .OR., .X | OR. | LNOT, LAND, LOR, LXOR LNOT, LAND, LOR SHR, SHL |
| **, .RES. | | | No translation |
| .EQ., .GT .UGT., .U =, >, < | | | = , |

4.3 Flag Tables

The following list shows all the Z-80 instructions that fit into a particular flag-fixup sequence. Each of the six flags are assigned a fixup action. Therefore, the six fixup actions determine the code that will appear for an equivalent translation of a given instruction. Any of the fixup actions (or flags) can be set to a "don't care" state; in this case, the fixup action will not generate any extra code during the translation of the instruction. The "FLAG" directive is used to control the Care or Don't Care setting of the six flags (see directive description for defaults).

NOTE: "-16" concantenated to an opcode refers to a 16-bit operation.

| | FLAGS | | | | | |
|-----|--|-------|-------|-------|-------|-------|
| | C | Z | S | P/V | N | H |
| 0 - | unaff | unaff | unaff | unaff | unaff | unaff |
| | call di djnz ei ex exx | | | | | |
| | halt in_(n) jp jr ld nop out (n) | | | | | |
| | out_(c) pop push res ret reti retn rst set | | | | | |
| 1 - | unaff | save | save | fix | reset | reset |
| | ldd don ldi | | | | | |

| | | C | Z | S | P/V | N | Hidel |
|---|---|----------------------------|--------|--------|--------|-------|-------|
| 2 | - | unaff | save | save | reset | reset | reset |
| | | lddr ldir | | | | | |
| 3 | - | save | affect | fix | affect | set | nofix |
| | | cpd cpi | | | | | |
| 4 | - | affect | affect | affect | affect | reset | nofix |
| | | adc_16 adc add | | | | | |
| 5 | - | affect | affect | affect | affect | set | nofix |
| | | sbc_16 sbc sub | | | | | |
| 6 | - | unaff | save | save | save | save | save |
| | | dec_16 inc_16 | | | | | |
| 7 | - | affect | save | save | save | reset | reset |
| | | rla rlca rra rrca | | | | | |
| 8 | - | affect | affect | affect | fix | reset | reset |
| | | rl rlc | | | | | |
| | | rr | | | | | |
| | | rrc | | | | | |
| | | sra rld | | | | | |
| | | rrd srl | | | | | |
| 9 | _ | save | affect | fix | reset | set | nofix |
| | | cpdr cpir | | | | | |

| | С | Z | S | P/V | N | H |
|------|-----------------|--------|--------|--------|-------|-------|
| 10 - | reset | affect | affect | affect | set | set |
| | and | | | | | |
| 11 - | reset | affect | affect | affect | reset | reset |
| | or | | | | | |
| 12 - | affect | affect | affect | affect | set | nofix |
| | cp neg | | | | | |
| 13 - | unaff | affect | affect | affect | reset | nofix |
| | inc | | | | | |
| 14 - | unaff | affect | affect | affect | set | nofix |
| | dec | | | | | |
| 15 - | affect | save | save | save | reset | unaff |
| | add_16 | | | | | |
| 16 - | affect | affect | fix | fix | unaff | nofix |
| | daa | | | | | |
| 17 - | unaff | save | | | | set |
| | cpl | | | | | |
| 18 - | affect | | | | reset | |
| | ccf | | | | | |
| | set (not use | ed) | | | | |
| 20 - | unaff | affect | nofix | nofix | reset | set |
| | bit | | | | | |
| 21 - | unaff | | fix | fix | reset | nofix |
| | in_(c) | | | | | |

| | С | Z | S | P/V | N | Н |
|------|------------------------------|-------|-------|-------|-------|-------|
| 22 - | unaff | fix | nofix | nofix | set | nofix |
| | ind ini outd outi | | | | | |
| 23 - | unaff | set | nofix | nofix | set | nofix |
| | indr inir otdr otir | | | | | |
| 24 - | - unaff | unaff | unaff | unaff | reset | reset |
| | scf | | | | | |

The following algorithm is used to generate flag-fixup code for instruction translation.

- (1) Mask instruction flags with current flag mapping.
- (2) Save FLAGS in temporary_register if any flag is a "save" or "fix".
- (3) Build equivalent line(s) of source code.
- (4) Save FLAGS in temporary mask_register if any flag is an "affect" and any other flag is a "save" or "fix".
- (5) If (4), mask out flags in temporary_mask_register which are not "affect".
- (6) If (2), mask out flags in temporary_mask_register which are "affect" or "fix".
- (7) If (2) and (4), combine temporary_register and temporary mask register into temporary_register.
- (8) Generate instruction(s) to fix each flag which is a "fix".
- (9) If (2), restore FLAGS from temporary_register.
- (10) Reset any flag which is a "reset".
- (11) Set any flag which is a "set".

APPENDIX A

EXAMPLES

A.1 General Example - DEM01.S

This example contains a mixture of Z-80 instructions and assembler pseudo-ops in a typical program format. The example was translated using the default mappings. The ".S", ".CRIT" and ".Z8000.S" files are shown to illustrate translator usage.

*H THIS IS A TEST

```
EXTERNAL EX1 EX2 EX3 EX4 ; SEE FILE OUTSIDE.S
      GLOBAL GLB1 GLB2 GLB3
    GLB3 DEFB 0
    TNTI
         DEFB 1
    EOUI
         EOU OFFH
         EQU START+4
EQU 100
    EOU2
    EOU3
    INT2
         DEFB 2
          DEFS 80
    GLB1
          LD A.B
          ADD HL, BC
          T.D
              B.C
          AND
              (IX+80H) ;OFFSET FROM RECORD
          JP LAB1
    LAB1
          LD A,B
          CPTR
          LD B,C
          DT
          TM
          RET
         STARTS ROUTINE TWO
CALL PROCESS
CALL NZ, EX1
RLC H
    :THIS STARTS ROUTINE TWO
    INT3
    GLB2
          EXX
          LDI
LD (IY+START),GLB1
RET
          DEFS 256
    INT4
             A,B EDALA
    INT5
          LD
              HL, EX2
          LD
         DEFT 'TITLE''S : (seldom used)'
    INT6
```

DEM01.28000.S

DEMOI MOD MODULE

```
SHE SHE STITLE THIS IS A TEST THE SHEET OF SHEET SHEET
EXI LABEL
                                          EX2 LABEL
                                          EX3 LABEL
                                          EX4 LABEL !SEE FILE OUTSIDE.S!
                           GLOBAL
                                          GLB1 LABEL
                                          GLB2 LABEL
                            GLB3 BYTE :=0
                             INTERNAL
                             INT1 BYTE :=1
                             CONSTANT
                             EQU1 := %OFF
                             EQU2 := START+4
                             EOU3 := 100
                             INTERNAL
                             INT2 BYTE := 2
                             NAME 01 ARRAY [ 80 BYTE ]
                             PROC 01 PROCEDURE
                             ENTRY
                                                                LDB RHO RH1
                             GLB1:
                                                                LDCTLB RL4 FLAGS
                                                                ADD R3 R1
LDCTLB RH4 FLAGS
ANDB RH4 #%(2)00100000
ANDB RL4 #%(2)00011111
ORB RL4 RH4
                                                                 LDCTLB FLAGS RL4
                                                                 LDB RH1 RL1
ANDB RH0 %80(R13) !OFFSET FROM RECORD!
                                                                RESFLG C
JP LAB1
LDB RH0 RH1
LDCTLB RL4 FLAGS
CPIRB RH0 @R3 R1 Z
LDCTLB RH4 FLAGS
                                                                 RESFLG C
                             LAB1:
                                                                 ANDB RH4 #%(2)00011000
```

```
ANDB RL4 #%(2)00100111
ORB RL4 RH4
         LDCTLB FLAGS RL4
LDB RH1 RH1
DI VI
         !IM 2!
         RET
!THIS STARTS ROUTINE TWO!
INT3: CALL PROCESS

GLB2: JR Z $+4
CALL EX1
         JR Z $+4 CALL EX1
         RLB RH3 #1
EX R1 R9
         EX R1 R10
EX R3 R11
   LDCTLB RL4 FLAGS
LDIB @R2 @R3 Rl
         LDCTLB FLAGS RL4
         LDB START(R14) #GLB1
         RET
 END PROC 01
INT4 ARRAY [ 256 BYTE ]
 PROC 02 PROCEDURE
 ENTRY
        LDB RHO RH1
LD R3 #EX2
TNT5:
                      #EX2
END PROC 02
```

INT6 ARRAY [* BYTE] := '%17TITLE%OS : (seldom used)'

END DEMO1 MOD

DEMO1.CRIT

```
( C Z S PV N H ) - CARE CARE CARE DONT CARE DONT CARE
(280 - 28000)
B-RH1 C-RL1 D-RH2 E-RL2
H-RH3 L-RL3 A-RH0 F-RL0
BC-R1 DE-R2 HL-R3 AF-R0
IX-R13 IY-R14 SP-R15 BC'-R9
DE'-R10 HL'-R11 AF'-R8
*H THIS IS A TEST
   EXTERNAL EX1 EX2 EX3 EX4 ; SEE FILE OUTSIDE.S
   GLOBAL GLB1 GLB2 GLB3 SAR CAR BIGI
       DEFB 0
GLB3
      DEFB 1
INTl
EOU1
       EQU OFFH
             START+4
       EOU
EOU2
       EOU
             100
EOU3
INT2
       DEFB 2
        DEFS 80
GLB1
        LD A,B
        ADD HL, BC
       LD on B, Consummer and anya to yagaa
        AND (IX+80H) ; OFFSET FROM RECORD
              LAB1
        JP
LAB1
        LD
              A,B
        CPTR
        LD
             B,C
        DI
        TM
             2
*** OPCODE NOT TRANSLATABLE
        RET
:THIS STARTS ROUTINE TWO
        CALL PROCESS
INT3
        CALL NZ, EX1
GLB2
        RLC
              H
        EXX
```

LD1 LD

RET

(IY+START), GLB1

INT4 DEES 256

INT'5 LD

A,B and anaged ered almod ene ered programmer the L,EX2 moreover to selled pair of a red and a red LD

INT6 'TITLE''S : (seldom used)' DEFT

A.2 Flag Example - DOCFF.S

This example shows the flag fixes for an instruction given varying Care and Don't Care patterns. The fixup pattern number refers to the flag tables of Section 4.3. Only the flags requiring fixing are varied. Also note the use of directives in the source file.

--- l --- FIXUP PATTERN
%FLAG(C Z S PV N H ON)
L1 LDD ;S ON PV ON
%FLAG(S ON PV OFF)
L2 LDD ;S ON PV OFF
%FLAG(S OFF PV ON)
L3 LDD ;S OFF PV ON
%FLAG(S OFF PV OFF)
L4 LDD ;S OFF PV OFF

:--- 21 --- FIXUP PATTERN %FLAG(C Z S PV N H ON) Z ON S ON PV ON Il IN B,(C) %FLAG(Z OFF S PV ON) IN B,(C) ; Z OFF S PV ON I 2 %FLAG(Z S OFF PV ON) ; Z S OFF PV ON I3 IN B,(C) %FLAG(Z ON S OFF PV ON) I4 IN B,(C) ; Z ON S OFF PV ON %FLAG(Z S ON PV OFF) 15 IN B,(C) ; Z S ON PV OFF %FLAG(Z OFF S ON PV OFF) ; Z OFF S ON PV OFF I6 IN B, (C) %FLAG(Z ON S PV OFF) 17 IN B, (C) ; Z ON S PV OFF %FLAG(Z S PV OFF) 18 IN B,(C) Z S PV OFF

DOCFF.Z8000.S

DOCFF_MOD MODULE

```
!--- 1 --- FIXUP PATTERN!
INTERNAL.
 PROC 01 PROCEDURE
PROC 01 PROCEDURE
ENTRY
L1: LDCTLB RL4 FLAGS
LDDB @R2 @R3 R1 !S ON PV ON!
!FIXUP PV FLAG!
        TEST R1
CLRB RH4
        TCCB NZ RH4
RLB RH4 #2
ORB RL4 RH4
        LDCTLB FLAGS RL4
       RESFLG N H
      LDCTLB RL4 FLAGS
L2:
     LDCTLB RL4 FLAGS
LDDB @R2 @R3 R1 !S ON PV OFF!
LDCTLB FLAGS RL4
RESFLG N H
LDCTLB RL4 FLAGS
T. 3:
       LDDB @R2 @R3 Rl !S OFF PV ON!
        IFIXUP PV FLAG!
TEST RI
CLRB RH4
TCCB NZ RH4
RLB RH4 #2
ORB RL4 RH4
LDCTLB FLAGS RL4
RESFLG N H
       RESFLG N H
L4:
       LDDB @R2 @R3 Rl !S OFF PV OFF!
        RESFLG N H
!--- 21 --- FIXUP PATTERN!

I1: LDCTLB RL4 FLAGS

CLR R4

LDB RL4, RL1
        INB RH1 @R4 !Z ON S ON PV ON!
        !FIXUP Z FLAG!
       !FIXUP Z FLAG!
TESTB RH1
CLRB RH4
TCCB Z RH4
RLB RH4 #2
RLB RH4 #2
ORB RL4 RH4
```

```
TESTB RH1 2.00088.98000
            CLRB RH4
            TCCB MI RH4
            RLB RH4 #1
RLB RH4 #2
ORB RL4 RH4
!FIXUP PV FLAG!
            TESTB RH1
         CLRB RH4
TCCB PE RH4
RLB RH4 #2
ORB RL4 RH4
LDCTLB FLAGS RL4
RESFLG N
LDCTLB RL4 FLAGS
CLR R4
IDR RL4, RL1
IDR RL4, RL1
            CLRB RH4
I2:
           CLR R4
LDB RL4,RL1
INB RH1 @R4 !Z OFF S PV ON!
!FIXUP S FLAG!
TESTB RH1
CLRB RH4
TCCB MI RH4
RLB RH4 #1
RLB RH4 #2
ORB RL4 RH4
!FIXUP PV FLAG!
TESTB RH1
CLRB RH4
CLRB RH4
LCLRB RH4
IFIXUP PV FLAG!
TESTB RH1
CLRB RH4
CLRB RH4
TCCB PE RH4
RLB RH4 #2
ORB RL4 RH4
LDCTLB FLAGS RL4
RESFLG N
LDCTLB RL4 FLAGS
CLR R4
LDB RL4,RL1
INB RH1 @R4 !Z S OFF PV ON!
!FIXUP PV FLAG!
T 3:
             !FIXUP PV FLAG!
             TESTB RH1
CLRB RH4
             CLRB RH4

TCCB PE RH4

RLB RH4 #2

ORB RL4 RH4
            LDCTLB FLAGS RL4
RESFLG N
LDCTLB RL4 FLAGS
I4:
             CLR R4
LDB RL4,RL1
             INB
                         RH1 @R4 !Z ON S OFF PV ON!
             !FIXUP Z FLAG!
             !FIXUP Z FESTB RH1
```

```
TCCB Z RH4

RLB RH4 #2

RLB RH4 #2

ORB RL4 RH4
!FIXUP PV FLAG!
         TESTB RH1
CLRB RH4
TCCB PE RH4
RLB RH4 #2 RH4
ORB RL4 RH4
         LDCTLB FLAGS RL4
         RESFLG N
T5:
         LDCTLB RL4 FLAGS
         CLR R4
         LDB RL4,RL1
INB RH1 @R4 !Z S ON PV OFF!
!FIXUP Z FLAG!
         TESTB RH1
         CLRB RH4
TCCB Z RH4
RLB RH4 #2
RLB RH4 #2
ORB RL4 RH4
         !FIXUP S FLAG!
         TESTB RH1
         CLRB RH4
TCCB MI RH4
RLB RH4 #1
RLB RH4 #2
ORB RL4 RH4
         LDCTLB FLAGS RL4
         RESFLG N
         LDCTLB RL4 FLAGS
I6:
         CLR R4
         LDB RL4,RL1
INB RH1 @R4 !Z OFF S ON PV OFF!
         !FIXUP S FLAG!
         TESTB RH1
         CLRB RH4
TCCB MI RH4
RLB RH4 #1
RLB RH4 #2
ORB RL4 RH4
         LDCTLB FLAGS RL4
         RESFLG N
         LDCTLB RL4 FLAGS
I7:
         CLR R4
         LDB RL4,RL1
INB RH1 @R4 !Z ON S PV OFF!
         !FIXUP Z FLAG!
TESTB RH1
                  RH4
         CLRB
```

```
Z RH4
RH4 #2
RH4 #2
RL4 RH4
FLAGS RL4
N
R4
     TCCB Z RH4
RLB RH4 #2
     RLB
     ORB
     LDCTLB
     RESELG
18:
     CLR
           R4
RL4, RL1
     LDB
           TNB
     RESFLG N
END PROC 01
END DOCFF MOD
```

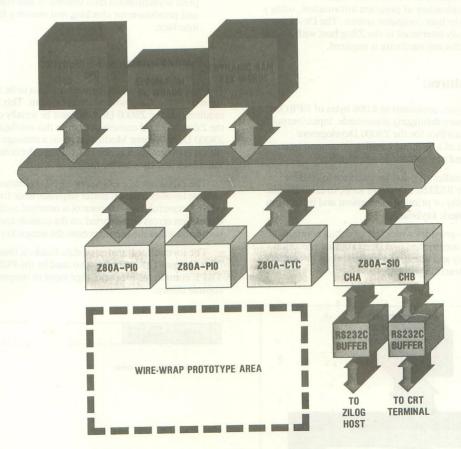
Z8000 DEVELOPMENT MODULE Timing and Logic Diagram

Z8000 DEVELOPMENT MODULE
Timing and Logic Diagram



Z8000™ **Development Module**

Product Brief August 1979



Z8000 Development Module Block Diagram

Features:

- Z8000 Microprocessor
- 2K words EPROM (expandable to 8K)
- 16K words Random Access Memory (expandable to 32K)
- Dual serial interfaces (110–19.2K Baud)
- 32 programmable I/O lines with handshake controls
- Four programmable 8-bit counter/timers
- Wire wrap area (18 sq. inches)
- Jumper-selectable CPU clock rates

Description:

The Z8000 Development Module is a complete, single board Z8000 microcomputer system specifically designed to assist the user in evaluating and developing hardware and software for Z8000-based products. It contains the Z8002 micro-

processor, 16K words of dynamic RAM, 2K word monitor PROM, dual serial interfaces, four counter/timers and 32 programmable parallel I/O lines. The memory resources of the board may be easily expanded with the addition of 16K RAM and 2K EPROM components. A wire wrap area also allows for addition of custom interfaces or special applications circuits.

The Z8000 Development Module is a high-performance prototype system with designed-in flexibility to accommodate a wide range of user applications. A variety of jumper areas and switches permit:

- Selection of 2.5 or 3.9 MHz clock rates
- □ Use of 2708, 2716 or 2732 EPROM's
- ☐ Use of 4K or 16K Random Access Memories
- Serial interface to modem, terminal or teletype
- □ I/O port addressing
- □ Baud rate selection (110–19.2K Baud)

General:

The Z8000 Development Module communicates with the outside world using the two RS232C serial interfaces. For standalone operation a single jumper-selectable interface for RS232C or current loop is used to communicate with a console device. The other serial interface may be connected to a Zilog Microcomputer or Development System to allow downloading and uploading of program information, using a utility program in the host computer system. The Development Module is easily interfaced to the Zilog host with a cable connection; no additional hardware is required.

Software Features:

The monitor program, contained in 4,096 bytes of EPROM, provides the necessary debugging commands, input/output control and host interface for the Z8000 Development Module. It consists of a terminal handler, command interpreter, debugger and upload/download handler.

- Terminal Handler provides interface to console device, either RS232C or current loop, to facilitate output to a display or printing mechanism and input from an alphanumeric keyboard.
- Debugger provides a basic set of debug commands to allow the user to start and stop program execution easily, display and alter CPU register, flags or memory, and trap instruction sequences.

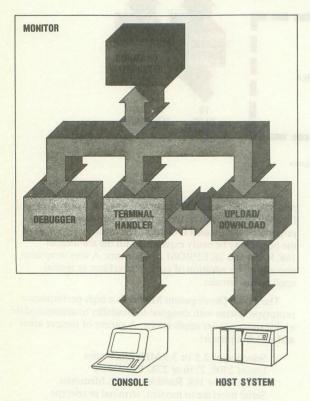


Fig. 1. Monitor Block Diagram

- Command Interpreter scans console inputs, insures command validity and passes requests to other software modules in the 4K monitor.
- Upload/Download Handler provides an interface between the serial connection and the host computer, the command interpreter and the memory resources of the Z8000 Development Module. It formats and interprets asynchronous data streams to and from the host and provides error checking and recovery for the serial interface.

Upload/Download Feature:

The upload/download feature allows data to be transferred from or to the Zilog host computer system. This feature requires that the Z8000 Development be serially connected to the Zilog host and console device. In this configuration the Z8000 Development Module serves as a message switcher and sits on the communications link between the console for the Zilog host and the console itself.

The execution of a LOAD or SEND command from the console invokes a data transfer sequence to or from the host. After successful transfer, control is returned to the console. Checksum errors are reported on the console and the transfer is aborted when the user depresses the escape key.

The format for transferred data blocks is illustrated in the figure below. This format is also used by the PUNCH and TAPE commands for paper tape input or output via the console device.

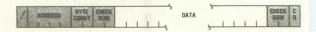


Fig. 2. Serial Data Format

Z8000 Development Module (Board size: 27.9×35.6 cm)

16K WORDS OF RAM 2K WORDS OF EPROM **EXPANDABLE TO 32K WORDS** EXPANDABLE TO 8K WORDS 4K AND 16K MEMORY COM-POWER CONNECTOR 2708, 2716 AND 2732 PONENTS ARE SUPPORTED **EPROMS ARE SUPPORTED** +127 **DUAL SERIAL INTERFACES** Z80-SI0/2 **SELECTABLE DATA RATES** Z80A-CTC = 20MA CURRENT LOOP RS232C CONNECTOR (TO/FROM HOST) RS232C/20 MA **CURRENT LOOP CONNECTOR** (TO CONSOLE DEVICE) I/O INTERFACE LOGIC SELECTABLE ADDRESS Commission of the Commission o RANGES **INTERFACES Z80A** PERIPHERALS TO Z8000 CPU BUS PARALLEL I/O 32 PROGRAMMABLE I/O LINES HANDSHAKE CAPABILITY WIRE WRAP AREA **Z8000 MICROPROCESSOR CPU BUS INTERFACE** RESET SWITCH Z8001 AND Z8002 **BUFFERED Z8000 SIGNALS** APPROX. 18 SQ. INCHES USER APPLICATIONS SUPPORTED **AVAILABLE USING J01** DEVELOPMENT JUMPER SELECTABLE CLOCK **AVAILABLE TO WIRE WRAP NON-MASKABLE INTERRUPT** AREA SWITCH RATE

Command and Summary:

The following notation is used in the command description.

<> Enclose descriptive names for the quantities to be entered, and are not actually entered as part of the

command

[] Denote optional entries in the

command syntax.

Denotes "OR", eg. W B denotes that either W or B may be used

but not simultaneously.

BREAK < address > Sets and clears a breakpoint at a [<n>] given memory address. The

option < n > allows specification of the number of occurrences, where n is from 1 to 128. The

default is one.

COMPARE < address 1 > < address 2> < n>

Compares two blocks of memory data beginning with the addresses specified for < n > bytes, where n is from 1 to 128. Errors are reported

on the console device.

DISPLAY < address > $\langle n \rangle [L|W|B]$

Displays and modifies memory for <n> number of words or bytes. The optional entry allows data to be handled as bytes, words or long words. The default is

words.

FILL < address 1>

Stores the < word > from <address 2> < word> memory address 1 to and includ-

ing address 2.

GO Begins program execution at the address contained in the current PC; execution is resumed where it was last interrupted. All registers are restored prior to execution.

IOPORT < address > [W B]

Allows direct communications from the console to a selected I/O port. A word (W) or a byte (B) may be read from the selected port and a word or byte may be sent to the selected port; default is byte.

JUMP < address >

Unconditional branch to the specified address. All registers are restored prior to execution.

MOVE < address 1> < address 2> < n>

Moves contents of a memory block from source address < address 1 > to destination address < address 2> for < n> bytes.

NEXT[<n>]

Executes the next $\langle n \rangle$ machine instructions. < n > may be from 1 to 128. If n is omitted 1 is assumed.

PUNCH < address 1> < address 2>

Punches a copy of memory from address 1 to address 2 on paper tape on the console device. Automatically turns on punch and a null leader is created. Upload/Download section describes the tape format used.

OUIT

Places serial channels into transparent mode. The Z8000 Development Module must be connected to both the Zilog host and the console device, and the Development Module acts as a

message switcher.

REGISTER

[< register name >]

Allows examination and modification of Z8000 registers. 8-bit, 16-bit or 32-bit quantities may be selected by the appropriate register-naming conventions.

TAPE

Loads memory from paper tape via the console device. The Upload/Download section describes the tape format used.

Specifications:

CENTRAL

CPU: Z8002 **PROCESSOR**

Clock Rate: 2.5 MHz or 3.9 MHz

MEMORY ROM: 2K words 2716 (expandable to 8K words)

RAM: 16K words 6116 (expand-

able to 32K words)

INPUT/OUTPUT

Parallel: 32 lines (two Z80A-PIO's) Serial: Dual RS232C or RS232C and Current Loop (Z80A-SIO) Note: The user has access to all bus signals which allow custom system expansion into the wire wrap area or off-board.

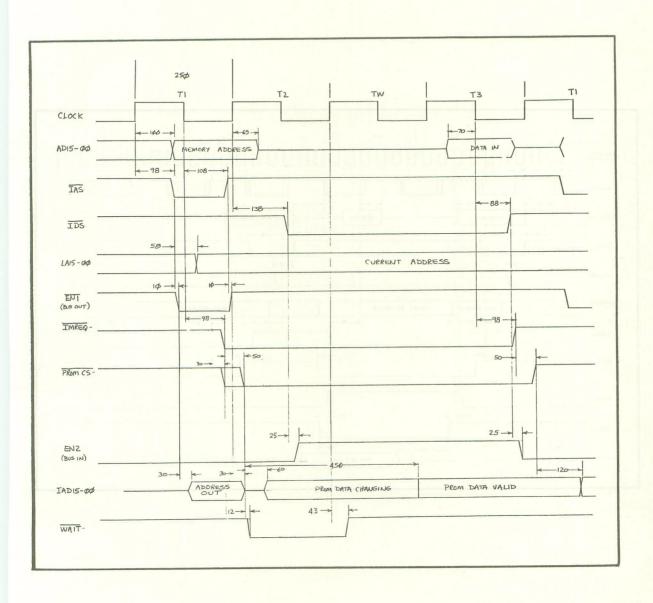
INTERRUPTS

Maskable Vectored (256) Maskable Non-Vectored Non-Maskable Segmentation Trap

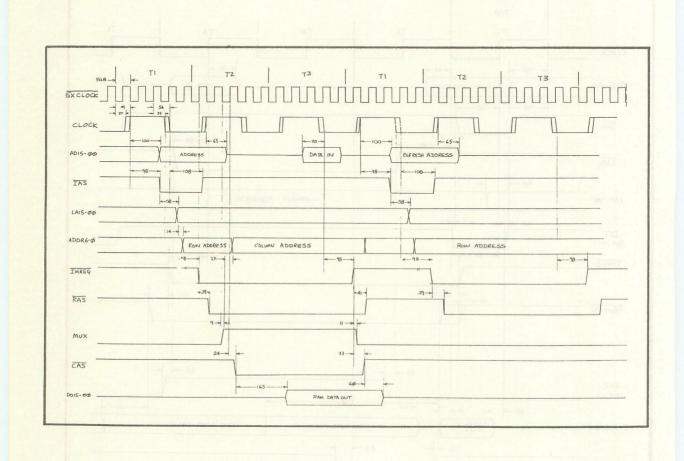
PHYSICAL **CHARACTERISTICS**

Height: 1.75 in.

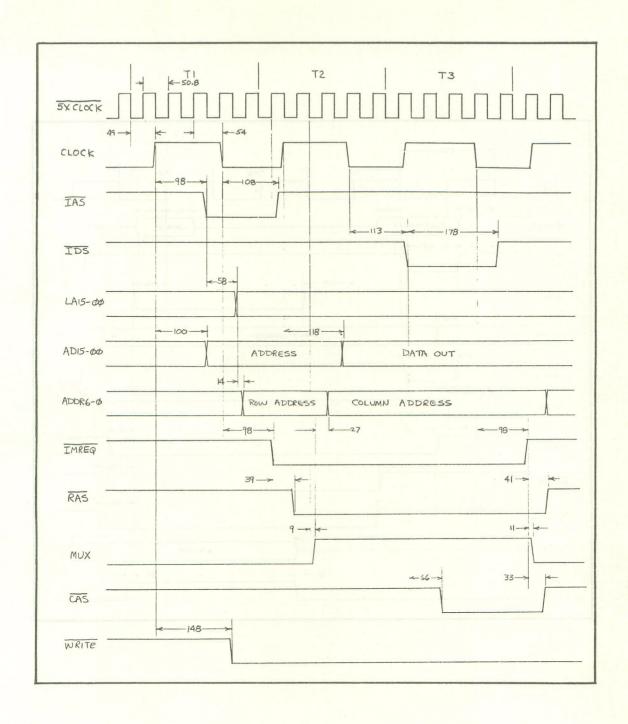
4.5 cm Width: 14.0in. 35.6cm Depth: 11.0in. 27.9cm Weight: approx. 30 oz. 850gm



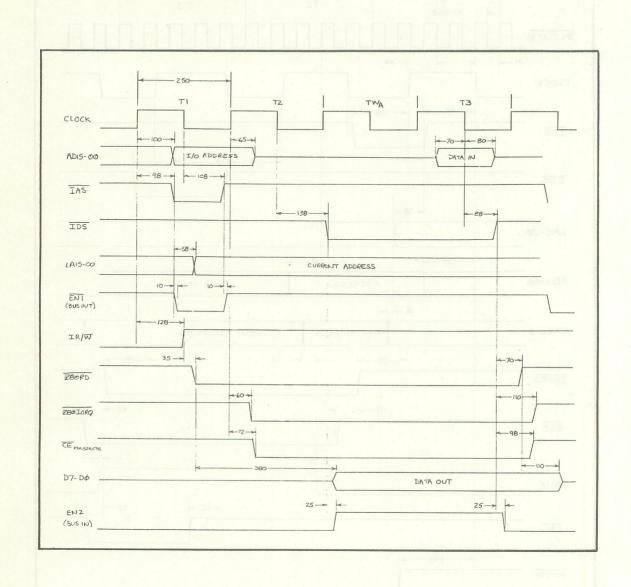
Sheet 1. PROM Memory Read Timing



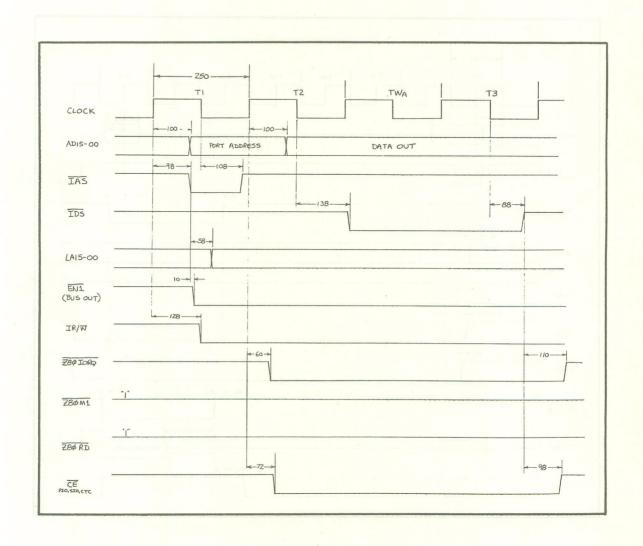
Sheet 2. Synchronous RAM Timing-Read Cycle, Refresh Cycle



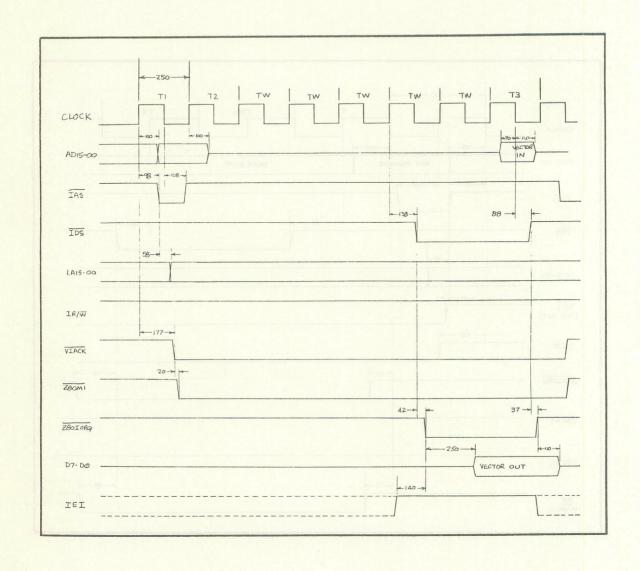
Sheet 3. Synchronous RAM Timing-Write Cycle



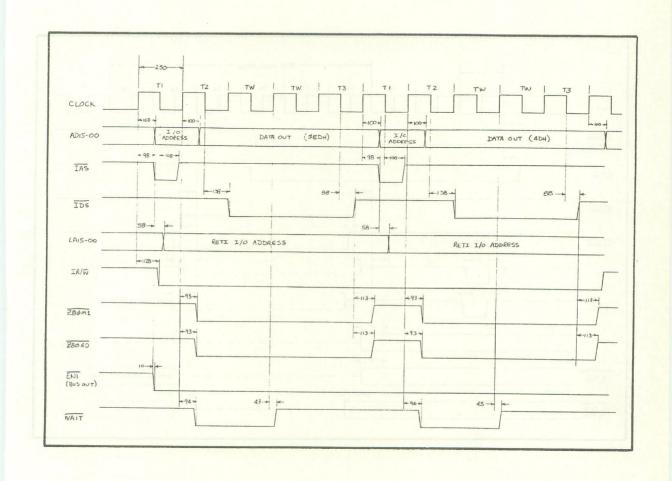
Sheet 4. I/O Read Timing



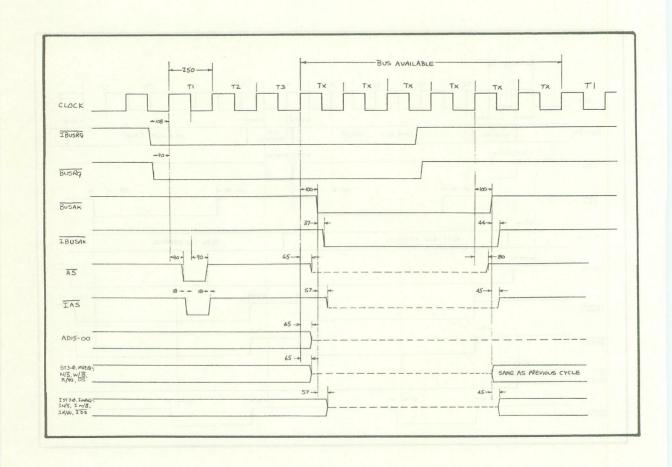
Sheet 5. I/O Write Timing



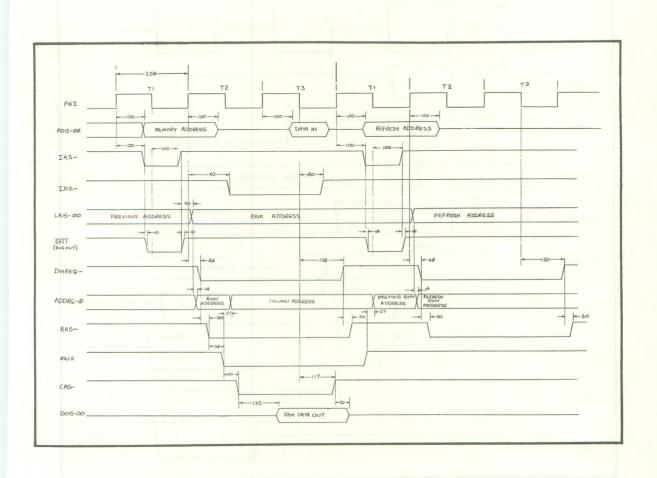
Sheet 6. Interrupt Acknowledge



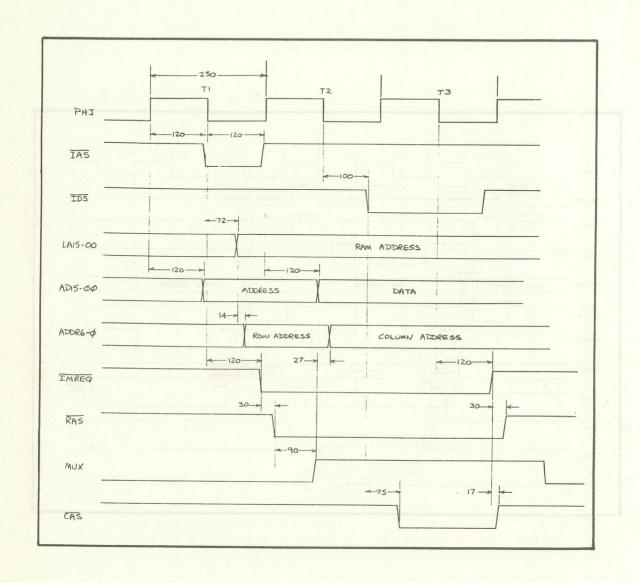
Sheet 7. RETI Cycle



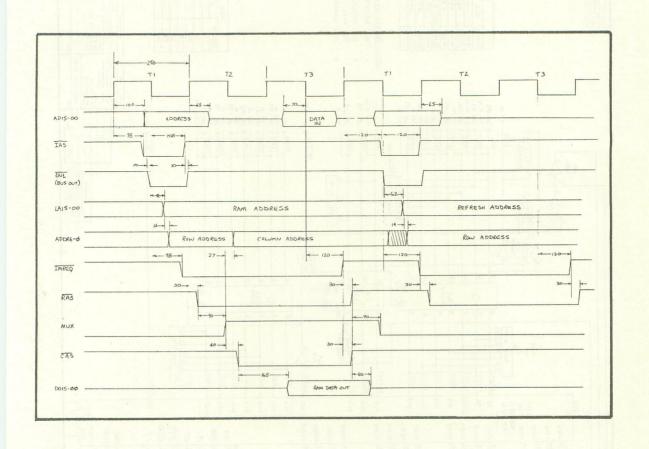
Sheet 8. BUSRQ/BUSAK Cycle



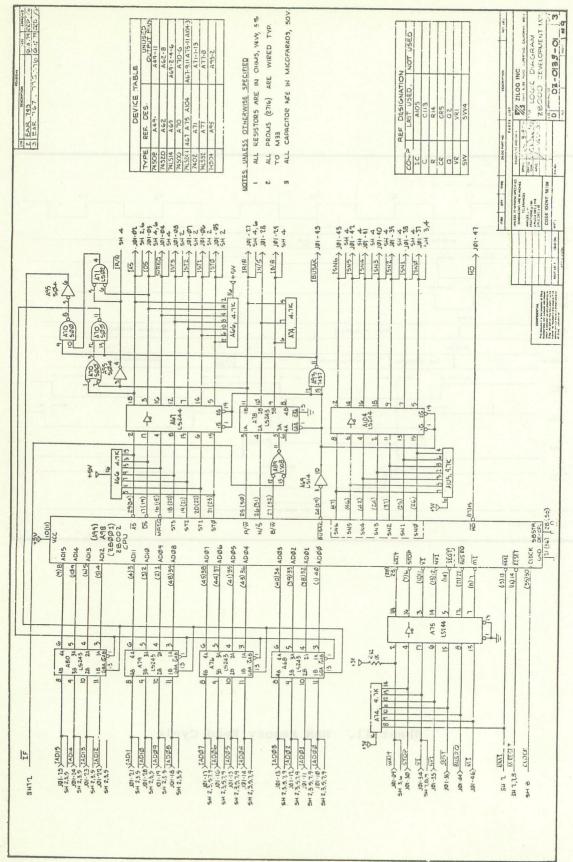
Sheet 9. RAM Timing - Read and Refresh Cycles
Using Delay Line

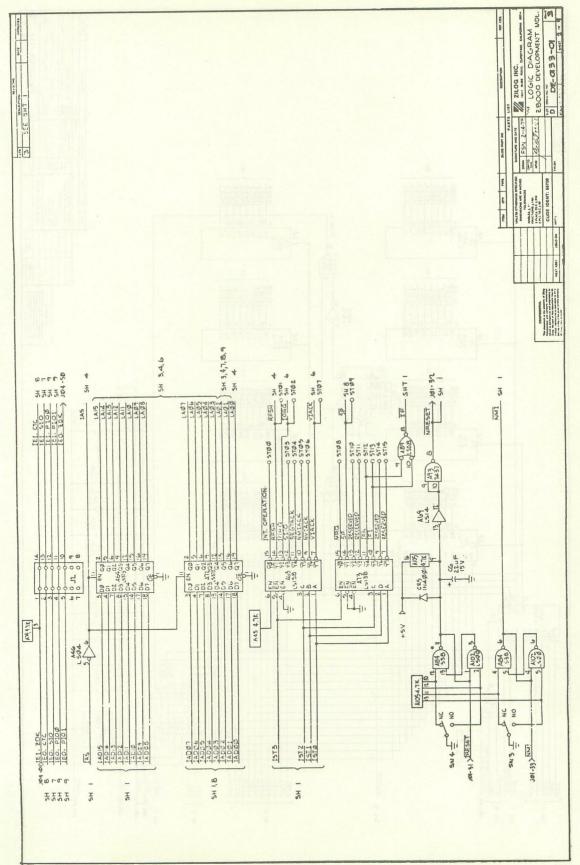


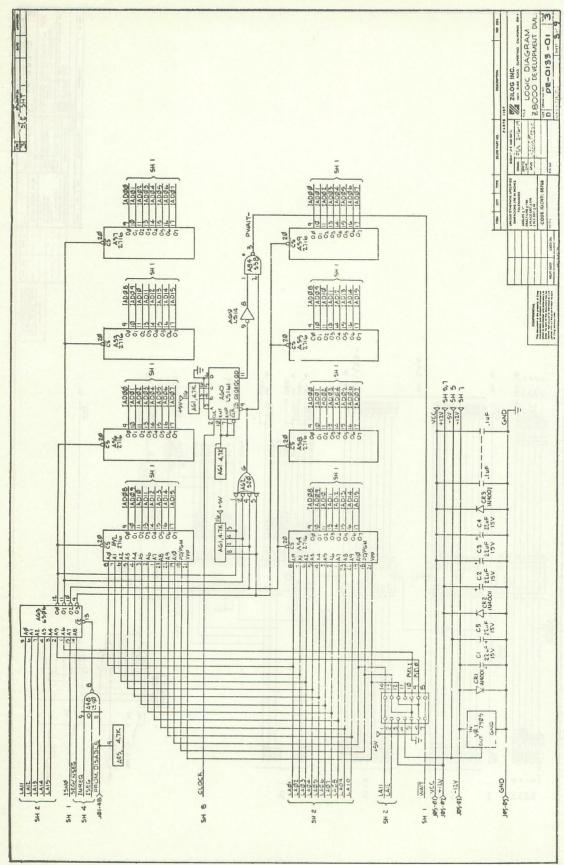
Sheet 10. RAM Memory Early Write Cycle

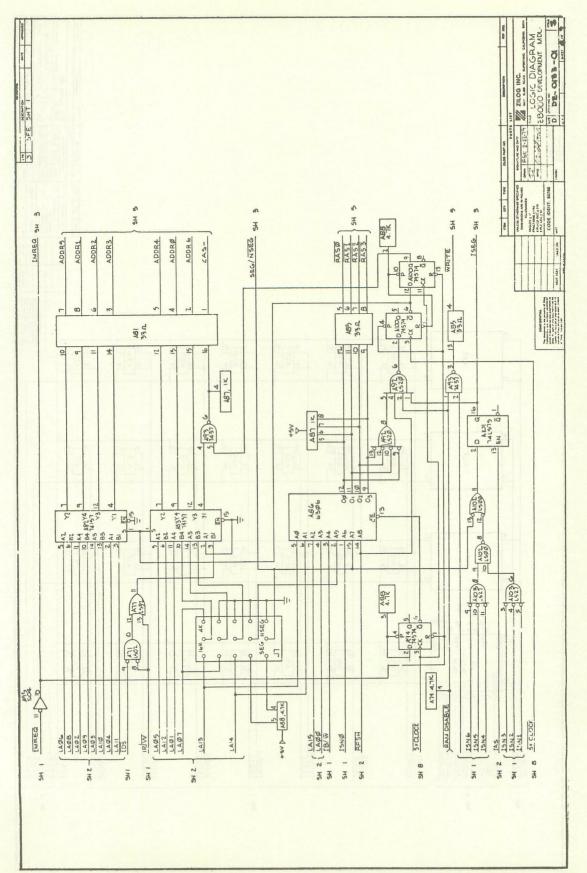


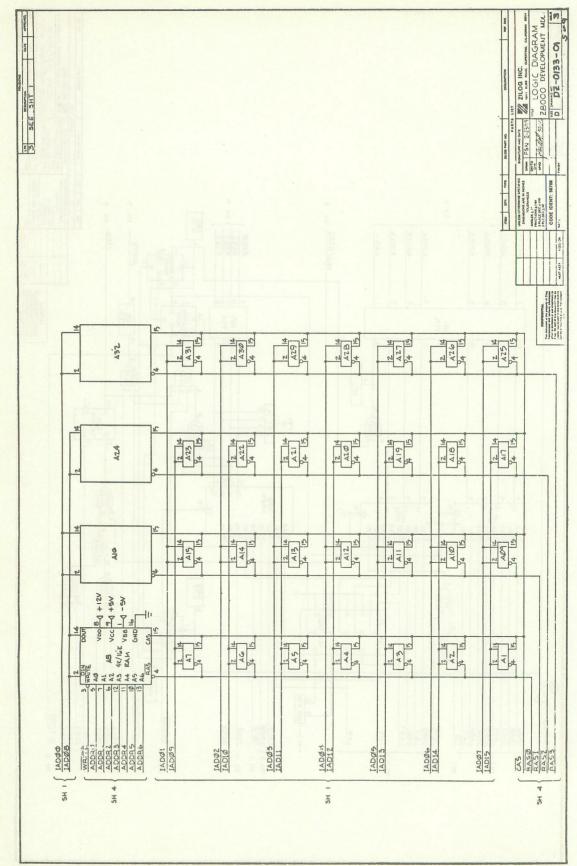
Sheet 11. RAM Memory Read Cycle

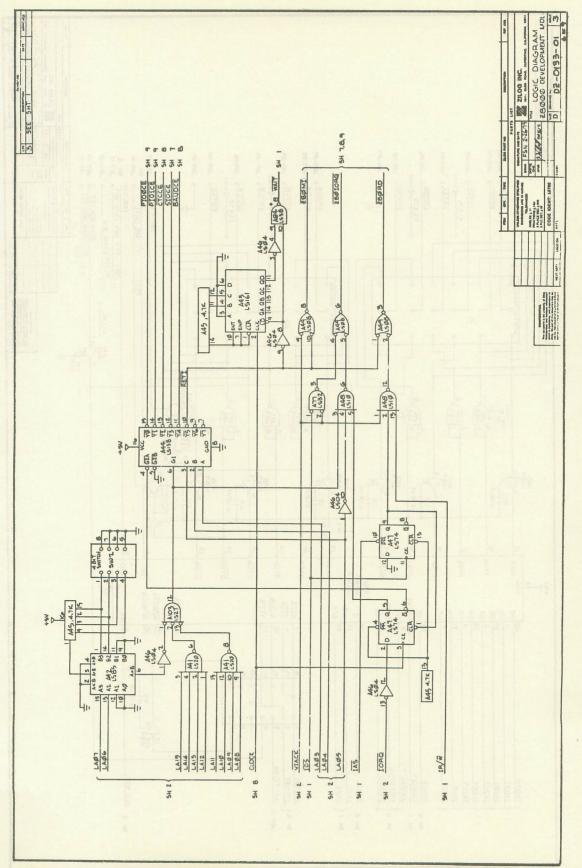


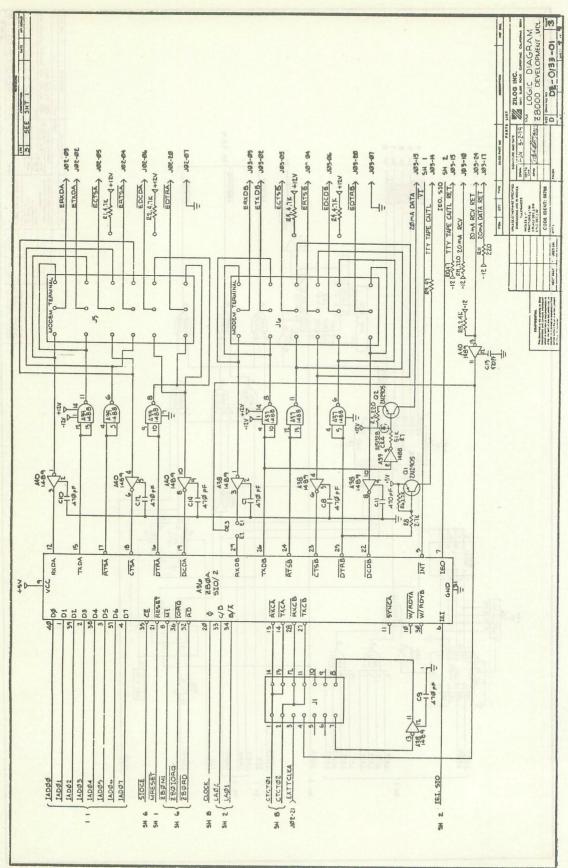


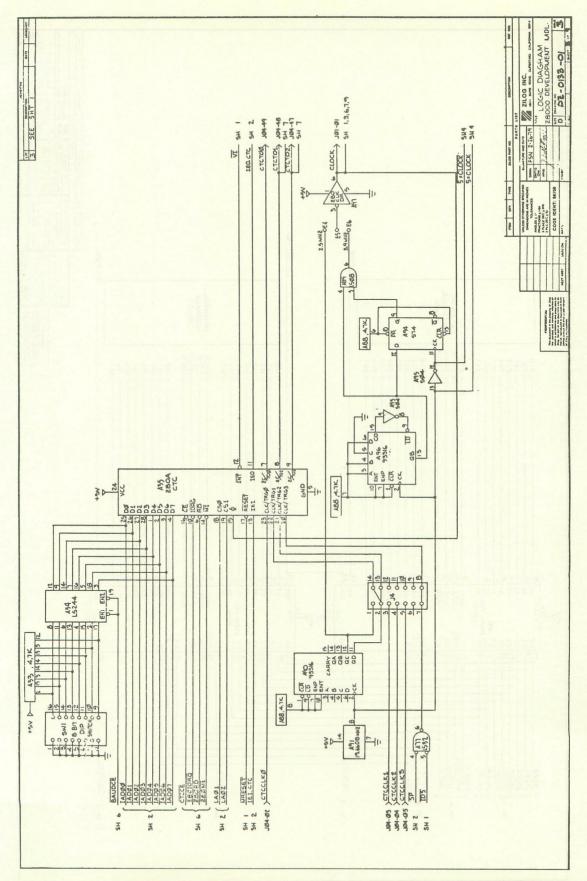


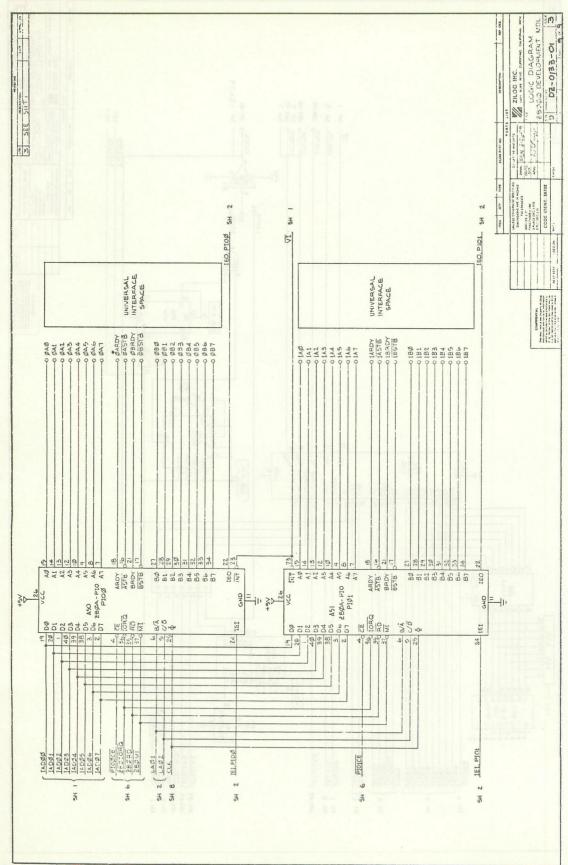












Z8000 DEVELOPMENT MODULE Monitor Program Listing

CONTENTS

| SECTION | 1 | INTRO | ODUCTION 46 | | | | |
|---------|---|-------------------------|-------------|-----|--|--|--|
| | | 1.1 | General | 469 | | | |
| SECTION | 2 | 1/0 | PROCEDURES | 470 | | | |
| | | 2.2 | General | 470 | | | |
| SECTION | 3 | MONITOR PROGRAM LISTING | | | | | |
| | | | General | | | | |

SECTION 1

INTRODUCTION

1.1 General

The Z8000 Development Module (DM) Monitor Program is provided on the Z8000 DM for use with the non-segmented Z8002 CPU. The program resides in 2K of EPROM on the board, starting at location 0, and uses an additional 576 words of dynamic RAM at locations %4000 through %447F for work space, program status area, and stack. This manual describes the most useful I/O procedures in the monitor program and contains a listing of the complete program. Detailed operating information for the monitor program is contained in the Z8000 DM Hardware Reference Manual.

SECTION 2

I/O PROCEDURES

2.1 General

The most useful I/O procedures are described and listed in this section. These procedures can be called upon by the user program to perform I/O functions through the console.

2.2 I/O Procedures

TYIN (%OFAO)

Get a character from the keyboard buffer. If the buffer is empty, this procedure waits for a character to appear. The character is stored in register RLO, and the contents of register RHO are destroyed.

TYWR (%OFC8)

Display a character in RLO to the terminal. The character will not be displayed if the XOFF character is received before this procedure is executed. This procedure waits until an XON character is received to display the character in RLO. If the display character is a carriage return, the zero flag is set and RHO is destroyed.

PUTMSG (%OFCO)

Send a character string to the terminal. Register $\Re 2$ contains the address of the character string buffer, and the first byte in the buffer is the number of characters to be displayed. If there is no carriage return in the string, the entire string specified will be displayed, otherwise the string will be displayed up to and including the first carriage return. Register contents $\Re 0$, $\Re 1$, and $\Re 2$ are destroyed.

TTY (%OFDC)

Receive and echo at the terminal a line of character string up to the first carriage return. The character string is stored in a buffer pointed to by register R2. Register R1 contains the size of the buffer. If the size of the character string received from the terminal exceeds the size of the buffer, the zero flag is set. All lower case alpha characters are converted to upper case characters before storing in the buffer. R1 returns the actual number of characters received from the terminal. The contents of registers R0 and R2 are destroyed.

CRLF (%OFD4)

Output a carriage return followed by a line feed to the terminal. Register contents RO are destroyed.

2.3 I/O Procedures Program

The following is a sample program to illustrate the I/O procedures introduced in this section. The progaram sends a message to the console to request an input which must match the string "Z8000". When the string is matched, a string of up to 20 characters is requested from the console for continuous display.

```
78000ASM 0.45
LOC OBJ CODE STMT SOURCE STATEMENT
                   1 FXAMPLES MODILLE
                   3 CONSTANT
                  5
                          ! SYSTEM EQUATES LINK TO Z8000 DM MONITOR PROGRAM!
                   6
                           ! INPUT/OUTPUT PROCEDURE !
                   7
                   8
                           TYME
                                  := Z0FC8
                  9
                           TYTN
                                := YOFA0
                  10
                           TTY
                                 := XOFDC
                           PUTMSG := ZOFCO
                  11
                  12
                           CRLF := XOFD4
                  13
                  14
                           ASCICR := XOD
                                               ! CARRIAGE RETURN !
                  15
                           BUFS17 := 20
                  16
                  17
                  18 INTERNAL
0000 OD 50 4C
                19 LODMSG ARRAY [14 BYTE] := [13,'P','L','E','A','S','F','',
0003 45 41 53
0006 45 20 4C
0009 4F 47 4F
000C 4F OD
                                        'L','O','G','O','N',ASCICR]
000E 13 57 48
                 21 MESMSG ARRAY [20 BYTE] := [19 ,'W','H','A','T',' ','I','S',
0011 41 54 20
0014 49 53 20
0017 54 48 45
001A 20 4D 45
001D 53 53 41
0020 47 45
                  22
                          1 1,1T1,1H1,1E1,1 1,1M1,1E1,1S1,1S1,1A1,1G1,1E1]
0022
                  24 BUFFER ARRAY [BUFSIZ BYTE] ! INPUT BUFFER!
0036
                  26 OUTBUF ARRAY [BUFSIZ+1 BYTE] ! OUTPUT BUFFER!
                  28
0000
                  29 MAIND PROCEDURE
                  30
                  31 ENTRY
0000 2102 0000'
                  32
                          LD
                                                ! SEND 'PLEASE LOGON' TO !
                                  R2, #LODMSG
                  33
                                                ! CONSOLE!
0004 5F00 OFC0
                  34
                          CALL
                                  PUTMSG
                  35
0008 C83A
                          LDB
                  36
                                  RLO,#1:1
000A 5F00 OFC8
                  37
                          CALL
                                  TYWR
                                               ! SEND CHARACTER : TO CONSOLE!
                  38
000E 2102 0022'
                  39
                          LD
                                  R2, #BUFFER
```

```
R1, BUFSIZ
0012 6101 0014
                  40 LD
0016 5F00 OFDC
                  41
                           CALL
                                   TTY
                                                 I ACCEPT INPUT FROM CONSOLE !
                  42
                                            ! INPUT LINE TOO LONG !
001A E6F2
                           JR
                                   Z, MAIND
                  43
001C 5402 0022'
                  44
                           I DI
                                   RR2, BUFFFR
0020 1002
                  45
                           CPI
                                   RR2,#178001
                                  NZ, MAIND ! INPUT MUST BE 'Z8000' !
BUFFER+4,#'0'
0022 5A383030
                  46
                           .IR
0026 FEEC
0028 4C01 00267
                  47
                           CPR
002C 3030
                  48
                           JR
002E FEER
                                   NZ, MAIND
                  49
                  50 REPEATS:
                                               I SEND 'WHAT IS THE MESSAGE' I
0030 2102 000E1
                  51
                          10
                                  R2,#MESMSG
0034 5F00 OFC0
                  52
                           CALL
                                  PUTMSG
                                                ! TO CONSOLE !
                  53
                  54
                                                 ! OUTPUT CARRIAGE RETURN & !
0038 5F00 0FD4
                           CALL
                                  CRLF
                  55
                                                 I LINE FEED TO CONSOLE !
                  56
                  57 ! ACCEPT MESSAGE FROM CONSOLE AND OUTPUT THE MESSAGE !
                  58 | CONTINUOUSLY TO THE CONSOLE!
                  59
003C C83A
                  60
                           LDB
                                  RLO,#1:1
003E 5F00 0FC8
                  61
                           CALL
                                  TYWR
                                                OUTPUT CHARACTER ':' !
                  62
0042 2102 00221
                  63
                           LD
                                   R2, #BUFFER
                        LD
0046 6101 0014
                  64
                                   R1, BUFSIZ
                          CALL
                                                 I ACCEPT I INE FROM CONOLE !
004A 5F00 OFDC
                  65
                                  TTY
                           JR
                                  Z, REPEATS
                                                 I INPUT LINE TOO LONG !
004E E6F0
                  66
                  67
                                                ! # OF CHARACTER IN INPUT LINE!
0050 6E09 00361
                  68
                           IDR
                                   OUTBUE, RL1
0054 2103 00221
                  69
                         LD
                                   R3, #BUFFER
0058 2102 0037'
                  70
                           LD
                                   R2, #OUTBUF+1
005C BA31 0120
                  71
                           LDIRB @R2,@R3,R1
                                                ! MOVE MESSAGE TO OUTPUT BUF!
                  72
                  73 LOOP:
                                   R2, #OUTBUF
0060 2102 00361
                  74
                           LD
0064 5F00 OFCO
                  75
                           CALL
                                  PUTMSG
0048 EBFB
                  76
                           JR
                                  LOOP
                                                ! OUTPUT MESSAGE FOR EVER !
                  77
006A
                  78 END MAIND
                  79
                  80 END EXAMPLES
                  81
```

SECTION 3

MONITOR PROGRAM LISTING

3.1 General

The following is a complete listing of the monitor program. An explanation of each module of the program is contained in a starred (*) box in the module listing.

```
Z8000ASM 0.45
    OBJ CODE STMT SOURCE STATEMENT
INC
                    1 ZDISPLAY MODULE
                   3 $SECTION DATA
                    4 CONSTANT
                    5
                             RXRDY
                                            0
                                                   ! TTY STATUS !
                                     :=
                             TXRDY
                                            2
                    6
                                    1=
                    7
                             PARITY :=
                                            7
                                                   ! PARITY BIT 7 !
                   8
                             LOWCSE
                                            5
                                                   ! LOWER CASE !
                   Q
                  10
                             ! PORT ADDRESS !
                  11
                  12
                             SIODA
                                    :=
                                          ZFF19
                                                  .! SIO CHANNEL A DATA!
                  13
                             STODE :=
                                          %FF1B
                                                 ! SIO CHANNEL B DATA !
                  14
                             SIOCA
                                          %FF1D
                                   1=
                                                 ! SIO CHANNEL A CONTROL!
                  15
                             STOCE
                                    :=
                                          WFF1F
                                                 ! SIO CHANNEL B CONTROL!
                             CTCO
                                          %FF11
                  16
                                    :=
                  17
                             CTC1
                                    :=
                                          %FF13
                  18
                             CTC2
                                    !=
                                          %FF15
                                                   ! CTC 3 FOR INTERRUPT!
                  19
                             CTC3
                                   1=
                                          %FF17
                  20
                             SWITCH :=
                                          %FF21
                                                   ! BAUDE RATE SETTING !
                  21
                             RETI
                                    :=
                                          %FF29
                                                  ! RESET CTC3 PORT ADDR !
                  22
                  23
                             COUNTER :=
                                          %47
                                                 ! COUNTER MODE FOR CTC!
                  24
                             CNTMOD :=
                                            %(2)11000111 ! COUNTER MDE CTC3 !
                  25
                  26
                  27
                  28
                                            %4460 ! USER PROGRAM AREA!
                             USERAM :=
                  29
                         1 ******************************
                  30
                  31
                                            %4400
                             PSAREA :=
                  32
                             PSBRK
                                            %4406 ! BREAK USE ILLEGAL INST.!
                  33
                                                   ! FOR TRAP!
                  34
                                                  ! NMI INTERRUPT !
                             PSNMI
                                    :=
                                            %4416
                  35
                             PSCTC
                                    :=
                                            %442A ! CTC INTERRUPT, VECTOR 6!
                  36
                             PSKBIN :=
                                            %4446 ! SIO CH.B, KB INTERRUPT!
                                                   ! VECTOR 14 !
                  37
                  38
                             PSCBRV :=
                                            %444A ! SIO CH.B SPECIAL REC. !
                  39
                                                    VECTOR 16 !
                  40
                             PSMZIN :=
                                            7.4456
                                                  ! SIO CH.A, MCZ INPUT !
                                                   ! VECTOR 1C !
                  41
                                            %445A ! SIO CH.A SPECIAL REC. !
                  42
                             PSCARV
                                   1=
                  43
                                                   ! VECTOR 1E !
                  44
                  45
                             VINT
                                            %(2)00010000000000000
                                                   ! VECTOR INT.BIT MASK !
                  46
                  47
                                                  ! FCW WORD !
                             BVINT
                  48
                                            12
                                                  ! BIT 12 FOR VECTOR INT. !
                  49
```

```
50
                             ESCHAR := %1B ! ESCAPE FOR ABORT LOAD/SEND!
                  51
                             ASCICE :=
                                            ZOD
                  52
                             ASCILF :=
                                            70A
                  53
                             BACKSP :=
                                            208
                  54
                             DELTE
                                            %7F
                                    !=
                  55
                             XOFFC
                                    1=
                                            713
                  56
                             XONC
                                     .-
                                            711
                  57
                             TAPOFF :=
                                            714
                  58
                             TAPON :=
                                            712
                  59
                             STACK
                                            %40A0 ! STACK POINTER !
                                    :=
                                                   ! SIZE OOAO BYTE !
                  60
                  61
                  62
                             CISIZE :=
                                                  ! CONIBE SIZE (BYTE)!
                                           128
                  63
                             COSIZE :=
                                           128
                                                 ! CONOBE SIZE(BYTE)!
                                            256
                  64
                             RBSIZE :=
                                                 ! RINGBF & MCZBUF SIZE!
                  65
                                           15
                  66
                             CMDNUM :=
                                                 1 # OF COMMAND !
                  67
                             ! BIT POSITION IN XOFFON !
                  48
                  69
                             TRNMDE :=
                                            0
                                                   I SET FOR TRANSPARENT MOFI
                  70
                             SNDMDE :=
                                            3
                                                  ! SEND COMMAND IN PROCESS!
                  71
                             LODMDE :=
                                                   ! LOAD COMMAND IN PROCESS!
                  72
                             INHOLD
                                                  ! SET KB INPUT BUFF 3/4 !
                                    :=
                  73
                                                   I FULL I
                  74
                             OUTHLD :=
                                                   ! SET IF XOFF IS RECEIVED!
                  75
                                                   FROM KEYROARD !
                  76
                             ESCAPE :=
                                            5
                                                   ! SET IF ESCAPE KEY HIT IN!
                  77
                                                   I LIPL DAD / DOWN DAD MODE !
                  78
                  79
                             XMSK
                                         %6
                                                  ! BIT MASK TO KEEP XON & !
                  80
                                                   ! XOFF BIT WHEN RETURN !
                                                   ! TO DEBUG!
                  82
                  83
                  84 $ABS %40B0
                  85
                  86 GLOBAL
40B0
                  87
                             CONIBF ARRAY [128 BYTE] ! BUFFER FOR INPUT CHAR!
                  88
4130
                  89
                             CONOBE ARRAY [128 BYTE] ! BUFFER FOR OUTPUT CHAR!
                  90
41B0
                  91
                             RINGBF ARRAY [256 BYTE] ! KB INTERRUPT INPUT !
                  92
                                                    ! BUFFER!
42B0
                  93
                             MCZBUF ARRAY [256 BYTE] ! INPUT FROM MCZ BUF !
                  94
43B0
                   95
                             CBUFADR WORD ! INPUT BUFFER ADDRESS !
43B2
                   96
                             CBUFSIZ WORD ! INPUT BUFFER SIZE !
                  97
                  98 ! RAM AREA NEED TO INITIALIZE WITH FIX VALUE !
                  99
43B4
                  100
                             INPTR WORD
                                               . ! CONIBF POINTER !
4386
                                               ! CONOBF POINTER !
                  101
                             OUTPTR WORD
```

```
43BB
                      CHRDEL BYTE ! CHAR DELETE ! LINE DELETE !
             102
43R9
             103
438A
             104
                      XONCHR BYTE ! XON CHAR !
43RR
             105
                      XOFCHR BYTE ! XOFF CHAR
43BC
             106
                      NULLCT NORD
                                    ! NULL CHAR
                      ILEGAL NORD ILLEGAL INSTRUC!
TIMONT HORD ! COUNT FOR CTC3 !
43BF
             107
43C0
             108
4302
             109
                      N_ WORD
                                 ! BREAK & NEXT COUNT !
             110
             111 ! RAM AREA NEED TO BE INITIALIZE WITH 7FRO VALUE !
             112
4304
             113
                      TTYPUT WORD | DEESET TO RINGRE FOR NEXT!
             114
                                   ! INPUT CHARACTER !
4306
             115
                      TTYGET WORD
                                    ! OFFSET TO RINGBE FOR NEXT!
             116
                                   ! FETCH CHAR !
4308
             117
                      MCZPUT WORD
                                  ! OFFSET TO MCZBUF FOR NEXT!
                                  ! INPUT FROM MCZ !
             118
43CA
             119
                                   ! OFFSET TO MCZBUF FOR NEXT!
                      MCZGET WORD
             120
                                   I CHAR TO OUTLIT TO TTY I
             121
             122
             123
43CC
                    INST WORD
                                   ! INSTRUCTION AT BREAK ADDR!
43CF
             124
                     BRKADR WORD
                                 ! VALUE OF R15 !
! USER FCW FOR BREAK & NEXT!
43B0
             125
                     RETSP WORD
43D2
             126
                     USFCH WORD
             127
4304
             128 CFLAGS WORD ! BIT 1 - INHOLD !
             129
                                    ! BIT 2 - OUTHLD !
       130 ! BIT 0 - TRANSPARENT MODE!
        ! UPLOAD/DNLOAD MODE!
133 ! BIT 4 - LOAD COMMAND IN !
             134
                                   1
                                          UPLOAD/DNLOAD MODE!
            136
137
             138
                RO_ HORD
R1_ HORD
R2_ HORD
         139
140
43D6
43D8
         141 R2_ WORD
142 R3_ WORD
143 R4_ WORD
43DA
43DC
43DF
43E0
      144 R5_
                           WORD
43F2
            145 R6_
                         WORD
            146 R7_ WORD
43E4
43E6
          147 R8_ WORD
        148 R9_ WORD
149 R10_ WORD
43E8
43EA
43EC
             150 R11_ WORD
43EE
            151 R12_
                           WORD
                           HORD
43F0
           152 R13_
                           WORD
43F2
           153 R14_
```

```
154 R15_ WORD
155 PC_ WORD
156 SAVFCH WORD
157 RETRY WORD
158
159 ! PAGE EJECT !
160
   43F4
   43F8
   43FA
                       161 $SECTION PRG1
                                      162
                         163 $ABS 0
                                       164
   0000 165 GLOBAL MAINP PROCEDURE
    166 ENTRY
167
   0000 0000 168 WVAL 0
   0002 4000 169 WVAL %4000 ! FCW VALUE !
                                                                                MSTART ! PC VALUE !
   0004 0010 170 WUAL
       171
   0006 4441 172 HVAL
                                                                               'DA'
  0008 5445 173 HVAL 'TE'
000A 3739 174 HVAL '79'
  000C 3231 175 WVAL '21'
000E 3520 176 WVAL '5'
177 178 MSTART:
                                      179
                               180 ! SET VALUES TO CTC 2 AND CTC 1!
                               181
   0010 8D38 182 CLR R3
  0012 3AB4 FF21 183 INB RL3, SWITCH ! BIT 7-4 CHANNEL A !
       184
                                                                                ! BIT 3-0 CHANNEL B !
   0016 A132 185 LD R2,R3
                                                                                                        ! SAVE SWITCH VALUE !
                     186
  0018 060B 0F0F 187 ANDB
001C 0103 0F38 188 ADD
0020 203C 189 LDB
                                                                               RL3, #XOF ! USE BIT 3-0 !
R3, #SWTBL ! GET VALUE FROM TABLE !
| 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 | 190 
                                                                                RL4, @R3 ! CTC VALUE FOR CH B !
                                                 OUTB
OUTB
                                                                               CTC1, RLO ! PROGRAM CHANNEL A !
                                                                               CTC1, RH4 ! CTC1 FOR SIO CHANNEL A !
  003A 3A46 FF13
                                       200
                                       201
                                        202 ! *** INITIALIZE SIO CHIP *** !
                                        203
                                                      LD R1, #INIOUT ! TABLE ADDRESS !
  003E 2101 0EC4
                                       204
  0042 2102 000B
                                                  LD
                                                                               R2, #11 ! SIZE OF TABLE !
                                       205
```

```
206 LD R3, #SIOCB
207 OTIRB @R3, @R1, R2 ! PROGRAM SIO CH. B!
0046 2103 FF1F
004A 3A12 0230
                   208
                  209 I D
004E 2101 OFDO
                                      R1, #MCZOUT
                  210 LD
211 LD
212 OTIRB
0052 2102 0009
                                      R2, #9
                                      R3, #SIOCA
0056 2103 FF1B
                                      @R3, @R1, R2 ! PROGRAM SID CH.A!
005A 3A12 0230
                  213
                  214
                  215 LD R1, #X9E00 ! RATE = F HEX !
216 LDCTL REFRESH, R1
005E 2101 9E00
                 215
0062 7D1B
                  217
                  218
                        ! INIT RAM DATA !
                  219
                  220
                 220
221 LD R0, #X4000
222 LD R1, #X4002 ! FILL RAM WITH 0 !
223 LD R2, #X200 ! FRUM 4000-4400 !
224 LD @R0, #0
225 LDIR @R1, @R0, R2
0064 2100 4000
0068 2101 4002
006C 2102 0200
0070 0D05 0000
0074 B801 0210
                 227 LD CONIBF, #' '
0078 4005 40R0
007C 2020
                228 LD RO, *CONIBF ! INIT CONIBF & CONOBF !
229 LD R1, *CONIBF+2
230 LD R2, *CISIZE ! 80 WORDS !
231 LDIR @R1, @R0, R2
007E 2100 40B0
0082 2101 4082
0086 2102 0080
008A BB01 0210
                  232
008E 2100 0EB4 233 LD RO, #RAMDTA ! RAM WITH FIX VALUE ! 0092 2101 43B4 234 LD R1, #INPTR
                       LDK
                 235
0096 BD28
                                    R2, #8
0098 BB01 0210
                 236
                             LDIR
                                      eR1, eR0, R2
                  237
                  238
                  239 ! PUT INTERRUPT ROUTINE ADDRESS IN PROGRAM STATUS BLOCK !
                LD RO, *PSBLK ! PROGRAM STATUS DATA !

242 LD R1, *PSAREA ! BLOCK DATA ADDRESS !

243 LD R2, *47 ! SIZE OF TABLE IN HORD !

244 LDIR @R1, @R0, R2
009C 2100 OEDA
00A0 2101 4400
00A4 2102 002F
00A8 BB01 0210
```

```
LD PSCBRV, R1 ! CONDITION!
00D8 6F01 444A
                                    257
                                    258 ASSESSED TO SEE THE SECOND SECOND
                                    259
                                                        LD R15, #STACK ! SET STACK POINTER !
00DE 210F 40A0
                                    260
                                    260 LD R15_/ R15
261 LD R15_/ R15
262 LB R1, #PSAREA
                                                                           R15_/ R15
00E0 6F0F 43F4
00E4 2101 4400
00EB 7D1D
00EA 7C05
                                    263 LDCTL PSAPOFF, R1 ! SET PS AREA POINTER !
                                                         EI
                                     264
                                     265
                                     266
267 END MAINP
                                     266
OOEC
                                     268 ! PAGE EJECT !
                                                      ! MAIN PROGRAM !
                                     270
                                     271
                                     272
                                     273 GLOBAL EROR LABEL
          274 GLOBAL DEBUG PROCEDURE
OOFC
                                    276 LD R1, *DEBUG ! PUSH RETURN ADDRESS!
277 PUSH @R15, R1
278 LD R1,CFLAGS ! CLEAR BIT EXCEPT XON/XOFF!
279 AND R1,*XMSK
00EC 2101 00EC' 276 LD
00F0 93F1
00F2 6101 43D4
00F6 0701 0006
00FA 6F01 43D4
                                    280 LD CFLAGS,R1
                                     281
                                    282
283 LDB
284 CALR
OOFE CB3C
                                                                           RLO, #1(1
0100 BCC6
                                                                           TYNR
0102 DCA2
                                     285 CALR GET ! GET 1 LINE INPUT,1ST !
                                     286
                                                            ! CHAR RETURN IN RLO!
                                     288 ! CHECK IF COMMAND IS VALID !
                                    289 LD
290 LDK
                                                                           R1, #CMDTBL ! COMMAND LITERAL TABLE !
0104 2101 0120
0108 BD2F
                                                                            R2, #CMDNUM ! # OF COMMANDS !
                               291 CPIRB RLO, QR1, R2, EQ
292 JR NZ, EROR ! IN
293 LDK R1, *CMDNUM-1
294 SUB R1, R2 ! IN
295 SLL R1, *1
296 ADD P1, *PONUTED
010A BA14 0286
010E EE1F
                                                                            NZ, EROR ! INVALID COMMAND !
0110 BD1E
0112 8321
                                                                           R1, R2 ! INDEX TO ROUTINE TABLE !
0114 B311 0001
0114 B311 0001
0118 0101 0130
                                    296 ADD R1, #ROUTBL
297 LD R3, @R1
011C 2113
                                                                           R3, QR1 ! GET ROUTINE ADDRESS !
011E 1E38 298 JP
                                                                            eR3
                                                                           'B'
'C'
'D'
'F'
                                     299
0120 42
                             300 CMDTBL: BVAL
                  301 BVAL
0121 43
                        302 BVAL
303 BVAL
0122 44
0123 46
                                                                           161
                      304
0124 47
                                                             BVAL
                                                                            111
0125 49
                                  305
                                                            BVAL
                                                                        1]1
1[1
1M1
                                              BVAL
                      306
307
0126 4A
                                              BVAL
0127 4C
                                              BVAL
                 308
0128 4D
```

```
0129 4F 309 RUAL 'N'
012A 50 310 BVAL 'P'
       311 BVAI 'P'
012B 51
0120 52
        312 RUAL 'R'
0120 53
       313 BVAL 'S'
012E 54 BVAL 'T'
012F 54 315 BVAL 'T' ! NEXT INST. AT EVEN ADDR!
     316
       317
     318
0130 0454' 319 ROUTBL: WVAL
                          BRKPNT
0132 01DE' 320 NVAL COMPARE
0134 021C' 321 NVAL DISPLAY
0136 019E' 322 NVAL FILL
0138 04BA 323 NVAL GO
013A 09BC' 324 NVAL IDPORT
013C 04AC' 325 NVAL JUMP
013E 0AB4' 326 NVAL LODCMD
0140 01C0' 327 NVAL MOVE
0142 0510' 328 NVAL NEXT
0144 0AOA' 329 NVAL PUNCH
0146 015E' 330 NVAL QUIT
0132 01DE' 320 NVAL COMPARE
014B 030E' 331 HVAL REGISTER
014A 0836' 332 HVAL SEND
014C 0A4E' 333 HVAL TAPE
014C 0A4E' 333 WVAL TAPE 334
     335
   337 1
            338 ! EROR RESET STACK POINTER, PUT OUT ? & JUMP TO DEBUG !
      339 1
            341 342 EROR:
342 EROR: 014E C83F 343 LDB RLO, **?' 0150 DCEE 344 CALR TYWR ! ? PUT ? FOR ERROR !
       345
0152 610F 43F4 346 LD R15, R15_ ! RESET STACK PTR ! 0156 4D05 43B6 347 LD OUTPTR, #CONOBF ! RESET CONOBF POINTER!
015A 4130
            348 JR DEBUG
015C E8C7
            349
            350 END DEBUG
            351 ! PAGE EJECT !
            352
015E
            353 GLOBAL QUIT PROCEDURE
            354
```

```
357 | DUIT ENTER THE TRANSPARENT MODE, ALL CHARACTERS !
              358 ! ARE PASSED BETWEEN THE MCZ & THE TERMINAL.
              359 ! FIRST CHECK ANY CHARACTER FROM TERMINAL , THEN
               360 ! X'MIT CHARACTERS TO MCZ UNTIL A CARRIAGE RETURN IS !
              361 | RECEIVED. AFTER THE CARRIAGE RETURN IS TRANSMITTED!
              362 ! TO MCZ, CHECK ANY CHARACTER FROM MCZ FOR OUTPUT TO !
               363 | TERMINAL, KEEP OUTPUT CHARACTER TO TERMINAL UNTIL !
               364 ! MCZ BUFFER IS EMPTY, THEN CHECK ANY INPUT FROM CRT. !
               365 1
               367
              368 ENTRY
              369 CLR
                             MCZPUT
015F 4D08 43C8
                       CLR MCZGET ! RESET MCZ INPUT BUF PTR!
0162 4D08 43CA
              370
              371 SET
0166 6500 43D4
                              CFLAGS, #TRNMDE! ENTER TRANSPARENT MDE!
              372 CRTIN:
              373 LD
016A 6100 43C6
                             RO, TTYGET ! GET INPUT FROM TTY!
              374 CP RO, TTYPUT
375 JR Z, MCZIN ! NO TTY
016E 4B00 43C4
0172 E608
                             Z, MCZIN ! NO TTY INPUT !
               376
              377 CALL GBFADR ! GET ADDRESS OF RINGBF!
378 LD TTYGET, RO ! UPDATE POINTER!
379 LDB RLO, QR2 ! GET CHAR FROM TTY!
0174 5F00 07DF1
0178 6F00 43C6
017C 2028
              380 CALL MCZSND ! & SEND TO MCZ !
381 JR NZ, CRTIN ! KEEP X'MIT DATA T
017E 5F00 078E'
                              NZ, CRTIN ! KEEP X'MIT DATA TO MCZ !
0182 EEF3
               382
                              ! TILL CR RECEIVE FROM TTY!
               383
               384 MCZIN:
0184 6100 43CA
               385 LD
                              RO, MCZGET ! GET INPUT FROM MCZ !
0188 4800 4308
              386 CP RO, MCZPUT
                        JR Z, CRTIN ! NO MCZ INPUT, GO & CHECK!
018C EAEE
               387
               388
                                       I TTY I
01BE 5F00 07F0' 389 CALL MCZADR ! GET ADDRESS OF MCZBUF!
               390 LD MCZGET, RO
391 LDB RLO, eR2 ! GET CHAR FROM MCZ !
0192 6F00 43CA
0196 2028
              392 CALL TYMR ! & SEND TO TTY !
0198 5F00 0776'
019C E8F3 393 JR MCZIN
                                       ! KEEP X'MIT DATA TO TTY !
                                       I UNTIL MCZBUF IS EMPTY!
               394
               395
               396 END QUIT
019E
               397 ! PAGE EJECT !
```

```
398
019F
              399 GLOBAL FILL PROCEDURE
              400
              401 1
              402 1
              403 ! FILL STORES GIVEN DATA (WORD) INTO SPECIFIED MEMORY I
              404 ! SPECIFIED ADDRESS MUST BE EVEN
              406 ! SYNTAX: FILL (BEGINNING ADDR) (ENDING ADDR) (DATA)!
              408 ! EXIT: JUMP BACK TO DEBUG, IF FOUND SYNTAX FREDR
                       RETURN TO DEBUG FOR NORMAL RETURN
              410 1
              412
              413 ENTRY
019F DC48
                      CALR LOD3 ! GET 3 ARGUMENT I
             414
01A0 F704
              415
                      JR C, CEROR ! NOT ENOUGH PARAMETERS!
              416
01A2 A750
             417 BIT
                          R5, #0
                                     ! CHECK FOR EVEN ADDR
                      JP NZ, EROR
            418
419
01A4 5F0F 014F
01A8 8354
                      SUB R4,R5
                                   ! ADDRESS2 - ADDREESS1 !
             420 CEROR:
                      JP C, EROR ! ERROR, ADDRESS2 TOO SMALL!
01AA 5E07 014E
              421
              422
            423 SRL R4, #1 ! FILL DATA IN WORD !
OIAE B341 FFFF
01B2 2F53
              424
                      LD @R5, R3 ! STORE DATA IN 1ST WORD !
01B4 9E06
              425
                      RET
                            Z
                                     ! R4-R5=0, JUST FILL 1 WORD!
              426
              427
             428
                      LD R3, R5 ! NEXT MEMORY ADDRESS !
01B6 A153
                      INC R3, #2
01B8 A931
01BA BB51 0430 430 LDIR @R3, @R5, R4 ! DUPLICATE MEMORY CONTENT!
01BE 9E08
                      RET ! RETURN TO DEBUG!
             431
              432
0100
             433 END FILL
             434
              435
              436
0100
              437 GLOBAL MOVE PROCEDURE
              440 1
             441 ! MOVE TRANSFER BLOCK OF DATA FROM ONE MEMORY TO
                                                     0840 7 10
             442 ! ANOTHER IN BYTE
             443 1
             444 ! SYNTAX: MOVE (SOURCE ADDR) (DEST. ADDR) N
             445 !
             446 ! EXIT: RETURN TO DEBUG
                                                     1
                                                     BOLA THE
             447 1
             448 | ******************************
             449
```

```
01C0 BC79
         453 JR C.CEROR ! NOT ENOUGH ARGUMENT!
01C2 E7E3
01C4 A156 454 LD R6, R5 ! CHECK DIRECTION TO MOVE!
01C6 8346 455 SUB R6, R4
01CB E703 456 JR C, MOVEUP
01CA BA51 0340 457 LDIRB @R4, @R5, R3
01CE 9E08 458 RET
459
                RET
01D4 8134 462 ADD R4, R3
01D6 AB40 463 DEC R4
01D8 BA59 0340 464 LDDRB @R4, @R5, R3
01DC 9E08 465 RET
O1DE 466 END MOVE
         447
    468
        469 GLOBAL COMPARE PROCEDURE
OIDE
          472 !
          473 | COMPARE TWO BLOCK OF MEMORY AND DISPLAY THEIR !
  474 ! CONTENTS IF THEY DON'T MATCH. N IS THE # OF BYTE !
  475 ! TO COMPARE.
 476 !
          477 ! SYNTAX: COMPARE (ADDR1) (ADDR2) N
    479 ! EXIT: RETURN TO DEBUG
          480 1
 481 ! *****************************
     482
         483 ENTRY
01EA 5E06 014E 490 JP Z, EROR
         491
01EE 2069 492 REPTS: LDB RL1, @R6 ! CONTENTS OF ADDR 1 !
         493 CPIB RL1, @R8, R7,EQ
01F0 BA80 0796
         494
01F4 F606
                JR Z, NXTSS
       495
01FC AB50 499 DEC R5
01FE DFFB 500 CALR PDIFF
0200 DC46 501 CALR PUTPTR
```

```
502
                 503 NXTSS:
                 504 TEST R7 ! CHECK IF ALL DONE!
0202 8D74
 0204 9E06
                 505
                           RFT
                                 7
                506
 0206 A960
                           INC
                                 R6, #1 ! NEXT MEMORY !
                507
508
 0208 E8F2
                           JR
                                  REPTS
                509 PDIFF:
020A DC6D
                510
                           CALR
                                  PUT2
                          LD
                                 R1, #1= 1
020C 2101 3D20
                511
                                 STRRG1
               512 CALR
0210 DFF9
0212 205D
                513
                                 RL5, eR5
                     1.08
                        INC
0214 DC70
                514
                                 PIIT1
                515
0216 6901 43B6
                                 OUTPTR, #2 ! ADD SPACE !
                516
021A 9E08
                          RET
                517
0210
                518 END COMPARE
                519 ! PAGE EJECT !
0210
                521 GLOBAL DISPLAY PROCEDURE
                522
                523 | *****************************
                524 1
                525 ! DISPLAY CONTENTS OF MEMORY AND ALSO ABLE TO ALTER
                526 ! ITS CONTENTS. IF (COUNT) IS GIVEN, 16 BYTES OF
                527 ! MEMORY ARE DISPLAYED ON ONE LINE, WITH SPACE BETWEEN !
                528 ! BYTE OR WORD OR LONG WORD DEPENDING ON THE LAST
                529 ! PARAMETER, IF (COUNT) IS NOT GIVEN, MEMORY IS GIVEN
                530 ! IN ONE BYTE/WORD/LONG WORD. NEW VALUE CAN BE ENTERED.
                531 ! OR CARRIAGE RETURN WILL ADVANCE TO NEXT MEMORY.
                532 | COMMAND D IS USED TO EXIT
                533 1
                534 ! SYNTAX:
                              DISPLAY (ADDRESS) [(COUNT)] [L:W:B]
                535 1
                536 ! EXIT:
                             IF 'Q' IS RECEIVED, RETURN TO DEBUG
                537 !
                              IF ANY ERROR IS FOUND, JUMP TO FRROR
                538 1
                539 ! ***************
                540
                541 ENTRY
                542
                543
                       ! IF NO INPUT AFTER DISPLAY COMMAND, DEFAULT TO !
                       ! DISPLAY MEMORY ONE WORD AT A TIME STARTING AT !
                    ! ADDRESS O !
                545
                546
                    CALR
021C DD6B
                547
                                 NXTARG
                                             ! SKIP OVER COMMAND
                548
                                              ! RETURN RLO=NEXT CHAR!
021E 8D38
                549
                        CLR
                                R3
                                              ! DEFAULT USE ADDRESS 0 !
0220 2107 0202
                550
                       LD R7, #%202
                551
0224 EE02
                552
                         JR
                                NZ, NOTCR
                                              ! THERE IS ARGUMENT !
                553
0226 A13B
                        LD
                                R11, R3
                                              ! GOT CR, DISPLAY ONE AT!
```

```
0228 E838 554 JR ONFMEN 1 A TIME 1
              555
              556 ! CHECK ANY ARGUMENT FOLLOW DISPLAY COMMAND & SPACE !
              557
              558 NOTCR: CALR GETADN ! NEXT ARGUMENT AS ADDR!
559 LD R11, R3 ! SAVE MEMORY ADDR!
560 JR Z, ONEMEM ! GOT CR, DISPLAY ONE AT!
022A DD85
022C A13B
022F F638
               561
                                        ! A TIME!
               562
                    ! NEXT ARGUMENT CAN BE (COUNT) OR (LIWIR) OR (CR)
                    ! IF (CR) IS RECEIVED, DISPLAY MEMORY ONE AT A TIME !
               565
                    ! IN HORD !
               566
0230 DEAD
              567 CALR CKWBL
                                         ! CHECK NEXT ARGUMENT !
                    JR Z, ONEMEM ! NO COUNT, RECEIVE LIWIB!
0232 E636
               568
               569
                                          ! DISPLAY ONE AT A TIME!
               570
                    ! NEXT ARGUMENT IS NOT (W/L/B) OR (CR), IT MUST BE !
               571
               572 ! COUNT. THE VALUE OF COUNT MUST > 0 !
               573
0234 DD8A
              574
                      CALR GETADN
                                         ! NEXT ARGUMENT MUST BE !
0236 A13D
              575 LD R13, R3 ! COUNT !
              576
0238 E602
                       JR Z, LINELP ! RETURN Z MEANS CR !
              577
                      ! AFTER COUNT, DEFAULT !
              578 ! TO DISPLAY WORD !
              579
              580 ! RECEIVE ADDR & COUNT, NEXT ARGUMENT CAN BE LIWIB!
              581 ! OR (CR). !
              582
                  CALR CKMBL ! GET (MIBIL) OR (CR) !
023A DFB2
              583
023C EE88
               584
                        JR NZ, EROR! SYNTAX EROR
               585
              587 ! PRINT MEMORY, 16 BYTES PER LINE WITH CORRESPONDING !
              588 ! ASCII CHARACTERS. R9 = # OF BYTE PER LINE (16) !
              589 ! RH7, RL7 = # OF BYTE PER SPACE (1/2/4) !
               590 ! R13 = ( COUNT ) R11 = ( ADDRESS )
              591 ! R8 = CONOBF ADDRESS FOR PRINTING ABSOLUTE VALUE !
              593 LINELP:
023E 8DD4
              594 TEST R13
              595 JR Z, EROR
0240 E686
              596
              0242 2109 0010
0246 A1B5
                                          ! CONVERT ADDR TO ASCII!
              599 CALR PUT2
0248 DC8C
                                         ! CHAR. & STORE CONOBF!
              600 INC OUTPTR
024A 6900 43B6
              601
              602 LD RB, #CONOBF+53
603 LDB @R8, #'*'
024E 2108 4165
0252 0C85 2A2A
0256 A980
              604 INC RB
              605
```

```
0258 208D 606 PUTIP: LDB RI5, 0R11
                                          GET MEMORY CONTENT !
025A 2E8D 607 LDB @R8, RL5 ! ABS VALUE IN BUF !
025C 0A0D 2020 608 CPB RL5, *' ! CK IF PRINTARIF
                        JR C, NOPRNT
0260 F703
               609
              610 CPB RL5, #%7F
611 JR C, PRNT
0262 0AOD 7F7F
                                           ! ONLY ALLOW 7 BIT VALUE!
                        JR C, PRNT
0266 E702
0268 0C85 2F2F
               612 NOPRNT: LDB
                            PR8, #1,1
                                           PLACE WITH .
         613
        614 PRNT:
026C A980
                        INC
                             RB, #1
                                           ! ADVANCE BUF ADDR !
026E 0C85 2A2A 615
                        INB
                              BRR, #7#1
                                           ! INCASE OF END
0272 NC9F 616
                        CALR PUT1
                                           ! CONVERT VALUE IN R5 !
               617
                                           ! TO ASCII&STORE CONOBF!
        618
                        TNC
0274 A9B0
                              R11, #1
                                           ! ADVANCE MEMORY ADDR !
        620
0276 AB90
                        DEC
                              R9
0278 FF08
                        JR
                              NZ, CKSP
                                         I CHECK IF FILL UP LINE !
027A A980 622 INC R8
027C 6F08 43B6 623 LD OUTPTR, R8 ! END OF BUF!
0280 DC86
         624 CALR PUTPTR ! YES, PRINT LINE !
                              RL7,RH7
0282 A07F
              625
                        LDB
                                         ! RESTORE # BYTE/SPACE !
626 !DJNZ R13 , NXTLNE CK IF ALL DONE !
0284 ABD0 627 DEC R13
              628
0286 EEDD
                        JR NZ, NXTLNE
               629
              630
0288 9F08
                        RET
              631
             632 CKSP: DBJNZ
028A FF1A
                             RL7, PUTLP
                                           ! CK IF NEED PUT SPACE !
028C 6900 43B6
             633
                        TNC
                              DITTETR
0290 A07F
              634
                        IDB
                              RL7, RH7 ! RESTORE # BYTE/SPACE !
                    ! DJNZ
DEC
                              R13 , PUTLP
              635
                                           IF NOT DONE, CONTINUE!
0292 ABD0
              636
                              R13
              637
                       JR
                              NZ, PUTLP
0294 EEE1
               638
                                           ! SAME LINE
                                                              ł
              639
              640 LD
0296 A980
                              R8
0298 6F08 43B6
                              OUTPTR, RB
                                           ! END OF BUF !
029C DC94
                              PHITPTR
               641
                    CALR
029E 9E08
               642
                        RET
               643
                                           ! RETURN TO DEBUG !
               644
               645
               646
                     ! PRINT MEMORY ONE AT A TIME IN BYTE/WORD/LONG WORD!
                    ! R9, R11 = MEMORY ADDRESS !
               647
                    1 RH7, RL7 = # BYTE/SPACE (1/2/4)
               648
               649
02A0 A1B9
              650 ONEMEN: LD
                              R9, R11
                                          ! SAVE ADDR FOR STOREING!
                              R5, R11 ! CONVERT ADDR TO ASCII!
PUT2 ! CHAR & STORE CONOBF!
02A2 A1B5
              651 LD
02A4 DCBA
               652
                      CALR PUT2
02A6 6900 43B6
             653
                        INC
                              OUTPTR
                                        ! ADD SPACE !
               654
02AA 20BD
              655 MORE: LDB
                              RL5, 0R11 ! GET MEMORY IN BYTE !
02AC DCBC
              656
                        CALR
                              PUT1
                                           ! CONVERT TWO 4BITS ABS!
```

| | | 657 | | | ! VALI | JE TO THO ASCII | [] |
|------|----------|---------|-------------|-----------------|-------------|-----------------|--------|
| 02AE | A9B0 | 658 | INC | R11, #1 | | MEMORY ADDR BY | 1! |
| | | 659 | | | | | 6 320 |
| 02B0 | FF04 | 660 | DBJNZ | RL7, MORE | ! CK IF | PRINT BINIL | ! |
| 02B2 | 6900 431 | 86 661 | INC | OUTPTR, #1 | ! ADD S | SPACE | 1 |
| 02B6 | A07F | 662 | LDB | RL7, RH7 | ! RESTO | DRE # BYTE/SPAC | 1 30 |
| | | 663 | | | | | |
| 02B8 | DFD9 | 664 | CALR | PGET | ! PRINT | CONOBF, GET IN | ITU9V |
| | | 665 | | | ! INTO | CONIBF, TRANSL | ATE! |
| | | 666 | | | ! INPL | IT TO DOUBLE WO | IRD! |
| | | 667 | | | ! HEX | VALUE IN R2, R3 | 3 ! |
| | | 668 | | | | | |
| 02BA | 9E07 | 669 | RET | C | ! RECE: | IVE QUIT! | |
| 02BC | E6F1 | 670 | JR | Z, ONEMEN | ! RECE! | IVE CR, ADVANCE | TO! |
| | | 671 | | | ! NEXT | LOCATION ! | |
| | | 672 | | | | | |
| | | 673 | ! RECEIVE \ | VALUE FOR MEMOR | RY, CHECK E | EITER REPLACE ! | A AUSO |
| | | 674 | ! CONTENT ! | ITH VALUE IN U | ONG WORD (| OR WORD OR BYTE | 100 |
| | | 675 | | | | | |
| 02BE | AAFO | 676 | DECB | RL7 | | | |
| 0200 | EE02 | 677 | JR | NZ, CKHL | ! CHECI | INPUT BINIL | |
| 02C2 | 2E9B | 678 | LDB | ery, RL3 | ! NEW I | BYTE CONTENT! | |
| 02C4 | E807 | 679 | JR | SRT | | | |
| | | 680 CKH | L: | | | | |
| 0206 | AAFO | 681 | DECB | RL7 | | | |
| 0208 | EE02 | 682 | JR | NZ, CKW | | | |
| | | | | | | | |

```
683 LD @R9, R3 ! NEW WORD CONTENT !
684 JR SRT
685 CKW:
02CA 2F93
02CC F803
02CF 2F92
           686 LD @R9, R2 ! NEW LONG WORD CONTENT!
                TNC
0200 4991
            687
                       R9, #2
           488 LD @R9, R3
02B2 2F93
            689 SRT:
       690 LDB RL7, RH7 ! RESTORE #BYTE / SPACE!
02D6 E8E4 691 JR ONENEM ! NEXT LOCATION !
            692
0208
            693 END DISPLAY
            694 AV CHARTER THOUSE ER CO STRUCTUO
            695
0208
            696 GLOBAL CKMBL PROCEDURE
     697
      700 ! CKMBL CHECK THE CHARACTER IN RLO WHETHER IT IS 'W' OR !
            701 ! 'B' OR 'L' OR CARRIAGE RETURN, THEN SET # OF BYTE TO !
            702 ! PRINT SPACE IN RH7, RL7
            703 1
            704 ! INPUT : RLO = CHARACTER
            705 1
        706 ! OUTPUT: RET Z & R7 = %0101 IF RECEIVING B
       707 ! RET Z & R7 = %0202 IF RECEIVING (W) OR CR !
     708 ! RET Z & R7 = %0404 IF RECEIVING L
                     RET NZ & R7= %0202 IF RECEIVING CHARACTER
            710 !
                                 OTHER THAN ABOVE
            711 !
            713
            714 ENTRY
            715 LD R7, #%202 ! DEFAULT IS 2 BYTE/' ' !
0208 2107 0202
            716 CPB RLO, #ASCICR
O2DC OAO8 ODOD
            717 RET Z ! RECEIVE CR, USE WORD !
02F0 9F06
            718
            719 CPB RLO, #'N' ! CK FOR W
02E2 0A08 5757
02E6 9E06 720 RET Z ! RECEIVE W!
           721
            722 LD R7, #X101 ! SET R1 TO 1 BYTE/' ! 
723 CPB RLO, #'B' ! CK FOR B !
02E8 2107 0101
02EC 0A08 4242
02F0 9E06
            724 RET Z
                                 ! RECEIVE B !
            725
02F2 2107 0404 726 LD R7, #%404 ! SET TO 4 BYTE
02F6 0A0B 4C4C
            727 CPB RLO, #'L' ! CK FOR L
02FA 9E06
            728 RET Z ! RECEIVE L!
            729 LD R7, $%202
730 RET
02FC 2107 0202
0300 9E08
                                 ! NOT CR, WIBIL
           731
            732 END CKHBL
            733
```

```
735 GLOBAL PGET LARFI
                                 736 GLOBAL PRIGET PROCEDURE
0302
                                  737
                                 739 1
                                 740 ! PRIGET CONVERTS R5 TO FOUR 8-BITS ASCII CHARACTERS
                                 741 ! AND STORE THEM IN CONOBF, THEN PRINT CONOBF WITHOUT !
                                 742 ! CARRIAGE RETURN, FETCH NEXT INPUT LINE AND STORE
                                 743 ! THEM IN CONIBF, CONVERT 8 INPUT ASCII CHARACTERS TO !
                                 744 ! 8 HEX VALUE INTO R2,R3
                                 745 1
                                 746 ! OUTPUT: R2,R3 = EIGHT 4-BITS HEX VALUE
                                 747 1
                                                          RETURN C FOR RECEIVING 'O'
                                                          RETURN NC, Z FOR RECEIVING CARRIAGE RETURN
                                                   RETURN NC, NZ FOR RECEIVING INPUT WITH CR
                                 750 1
                                  751 1
         752 ! *** RO,R1,R2,R3,R4 ARE LOST ***
       753 1
                                 754 ! некончинием принцинентини принцинентини принцинент принцинен
                                 756 ENTRY
0302 DCE9
                                                     CALR PUT2 ! PUT R5 INTO CONOBF!
                                 757
0304 6900 4386
                                 758 INC OUTPTR, #1 ! PUT SPACE AFTER!
                                 759 PGET: CALR PUTNCR ! OUPUT TO TTY W/O CR !
0308 DCC4
030A 5E08 0656'
                                 760 JP GETNUM! GET INPUT IN R2,R3, RET!
                                 761
030E
                                 762 END PRIGET
                                 763 ! PAGE EJECT !
                                 764
030E
                                 765 GLOBAL REGISTER PROCEDURE
                                 768 1
                                 769 ! REGISTER COMMAND DISPLAYS CONTENTS OF USER REGISTERS. !
                                 770 ! USER IS ABLE TO ALTER ITS CONTENT, IF NO REGISTER NAME!
                                 771 ! IS GIVEN, ALL REGISTERS WILL BE DISPLAYED IN WORD. !
                                 772 ! IF REGISTER NAME IS GIVEN, NEW VALUE OR CARRIAGE RETURN!
                                 773 ! CAN BE ENTERED AFTER THE CURRENT VALUE, THEN ADVANCE TO!
                                774 ! NEXT REGISTER.
                                                                                                            12107 6101 722
                                 775 1
                                 776 ! SYNTAX: REGISTER (REGISTER NAME)
                                 778 ! EXIT: IF Q COMMAND IS ENCOUNTERED, DIRECT JUMP BACK TO!
                                 779 | BEBUG, FIX STACK
                                 780 ! IF ERROR IS FOUND, DIRECT JUMP BACK TO DEBUG
                                                 OUTPUT '?' AND FIX STACK
                                 781 !
                                 782 !
                                 784
                                 785 ENTRY
030E DDE4
                                 786 CALR NXTARG ! SKIP OVER COMMAND
```

```
787
                         JP Z, RALL ! HIT CR,NO NAME, PRINT ALR!
0310 5E06 0436
                          CPB RLO, *'R' ! IF NAME ENTER, MUST BE R!
                788
0314 0A08 5252
0318 FF1D
                789
                          JR NZ, ERNZ ! ERROR!
                790
                791
                      ! CHECK REGISTER NAME FOR RR/RH/RL/R/ !
                792 ! RL6 = 'L' OR 'H' OR 'R' OR # !
                793 ! RH6 = O IF INPUT IS 'RL' OR 2 IF INPUT IS 'RH' !
                794
                795
                          CALR GCHAR ! GET NEXT CHAR
031A DD88
                               7, ERSS | ERROR TO RECEIVE CR |
031C E60D
                796
                         .IR
                         I DB RIA, RLO ! SAVE 1ST CHAR (R,L,H,#) !
031E A08E
                797
                       CPB RLO, #'R' ! CK IF DOUBLE REGISTER !
0320 0A08 5252
                798
                                          ! YES, GET REG #
0324 E608
                799
                          JR
                                 7. GDIG
                        CLRB RH6
                                           ! RH6=0 FOR RL#
                800
0326 8068
                          CPB RLO, #'L' ! CK LOWER BYTE REGISTER!
0328 0A08 4C4C
             801
032C E604
                802
                          JR
                                Z, GDIG
                                           ! YES, GET REG # !
                             RH6, #2
RLO, #'H'
                          LDB
                                          ! RH6=2 FOR RH# !
032E C602
                803
                                          ! CK UPPER BYTE REGISTER!
                          CPR
0330 0408 4848
                804
                          .IR
                                NZ, SAU1
                                           ! NO, RECEIVE WORD REG. NAME!
0334 EE03
                805
                806
                807 ! FETCH NEXT 2 CHARACTERS IN R1 FROM CONIBF, FILR 2ND!
                    ! CHARACTER WITH BLANK IF ONLY RECEIVE ONE CHARACTER!
                808
                809
                810 GDIG: CALR
0336 DDC6
                                 GCHAR
                                            ! GET 1ST CHAR OF REGNAME !
                811 ERSS: JP Z, EROR ! CAN'T BE CR !
0338 5E06 014E
                812
                813 SAV1: LDB
                                 RH1, RLO
                                          ! SAVE 1ST CHAR OF REG NAME!
033C A081
                          CALR
                                 GCHAR
                                          ! GET 2ND CHAR FOR REG NAME!
033E DDCA
                814
                                          ! SAVE 2ND CHAR !
                          JR
                                 NZ, SAV2
0340 FE02
                815
                816 LDB
                                 RL1, #' ' ! PUT ' ' IF HIT CR !
0342 C920
                817 JR
                               CKNAME
0344 E801
                818 SAV2: LDB RL1, RL0 ! BOTH CHAR IN R1!
0346 A089
                819
                820 ! CHECK REGISTER NAME IN R1 AGAINST REG LITERAL TBL!
                     ! IF FOUND MATCH, THEN !
                821
                822 ! R9 = PTR OF REG LITERAL TABLE NHICH MATCH THE NAME!
                823 ! R8 = OFFSET TO REG CONTENT ARRAY !
                824
                825 CKNAME: LD
                                 R8, #18
                                           ! TOTAL 18 REG NAMES !
0348 2108 0012
                826 LD R9, #LRO ! ADDR OF REG LITERAL TBL!
034C 2109 0E7C
                                 R1, 0R9, R8, EQ !LOOK FOR MATCH !
                827
                          CPIR
0350 RB94 0816
                828 ERNZ: JP
                                 NZ, EROR
0354 SEOE 014E
                                 R9, #2 ! BACK UP TO CORRECT ADDR!
                829
                          DEC
0358 AB91
                                 R2,#17 ! CONVERT REMAINING COUNT !
035A 2102 0011
                830
                         LD
                                 R2,R8 ! TO REG MEMORY CONTENT!
                831 SUB
035E 8382
                832 LD
                                 R8, R2 ! OFFSET!
0360 A128
                833 SLL R8, #1
0362 B381 0001
                834
                    ! DETERMINE WHAT TYPE OF REGISTER TO DISPLAY !
                835
                836
                                 RL6, #'H' ! RL6 = H/L/R/#!
0366 0A0E 4848
                837
                          CPB
                          JR
                                 Z, HALF
                838
036A E644
```

```
036C 0A0E 4C4C
             939 CPR RIS, #111
             840 JR Z, HALF
841 CPB RL6, ‡'R'
0370 FA41
0372 0A0E 5252
0376 E618
             842 JR Z, DWORD
             843
             844 ! DISPLAY WORD REGISTER. !
             845 ! R8 = OFFSET TO REGISTER CONTENTS !
             846 ! R9 = POINTER TO REGISTER LITERAL TABLE !
             847
0378 2101 5220
             848 RWORD; ID 81, #/8 /
037C DF9F
             849 CALR STREET
037E 6800 4386
             850
                     DEC OUTPTR, #1 ! PRINT ONLY R !
0382 2191 851 LD R1, @R9 ! PRINT REG NAME !
0384 DFA3
            852 CALR STRRG1
0386 6900 4386 853 INC OUTPTR, #1 ! SPACE AFTER NAME !
038A 6185 43D6 854 LD R5, R0_(R8) ! GET CONTENT!
         855
038F D047
         856 CALR PRIGET ! STORE RS IN CONOBE, OUTPUT!
857 ! CONOBF, EXPECT INPUT IN ! 858 ! R2,R3 !
          859
0390 9E07
                     RET C ! RECEIVE Q, RETURN TO DEBUG!
0392 FA07
       860 JR Z, NOINPUT ! RECEIVE CR, NO INPUT !
        861
0394 2192 862 LD R2,0R9 ! IF CHANGE PC, INPUT BE!
0396 0B02 5043 863 CP R2,#'PC' ! BE EVEN NUMBER!
       864 JR NZ,NOTPC
865 RES R3,#0
039A EE01
039C A330
           866 NOTPC:
039E 6F83 43D6 867 LD
                          RO_(RB), R3 ! REPLACE WITH NEW VALUE !
        868 NOINPUT:
03A2 A9B1
       869 INC
                         R8, #2 ! OFFSET+2 FOR NEXT CONTENT!
03A4 A991
                    INC R9, #2 ! PTR TO NEXT LITERAL!
            870
03A6 0B09 0E9F 871 CP R9, #LFC+1 ! DONE IF ADDR OF REG LIT.!
03AA E7E6
                    JR C, RWORD ! BEYOND THE TABLE !
             872
03AC 9E08
           873 RET ! RETURN TO DEBUG
             874
 875 ! DISPLAY DOUBLE WORD REGISTER, REGISTER NAME MUST BE!
             876 ! EVEN #, R8 = OFFSET TO REGISTER CONTENTS !
             877
                 ! R9=POINTER TO REGISTER LITERAL R1=REGISTER NAME !
            878
03AE 0B09 0E9A 879 DWORD: CP R9, #LR15 ! ONLY 0-14 (EVEN) !
       880 JR NC, EOR ! CAN'T BE PC,FC !
03B2 EF22
03B4 0A09 2020
            881 CPB RL1, *' ! CHECK IF 0-9 !
882 JR Z, CK1 ! YES, CHECK 1ST CHAR !
03B8 E602
03BA A710
            883 BIT R1, #0 ! CHECK 2ND CHAR IF EVEN!
03BC E801 884 JR NX
03BE A718
             885 CK1: BIT R1, #8 ! CHECK 1 CHAR IF EVEN!
03CO EEC9
             886 NX:
                    JR NZ, ERNZ ! ERROR FOR ODD REG NAME!
             887
03C2 2101 5252 888 DOUBL: LD R1, #'RR'
03C6 DFC4 889 CALR STRRG1
0308 2191
       890 LD R1,0R9 ! PRINT REG NAME!
```

```
03CA DFC6 891 CALR STRRG1
03CC 6900 43B6 892 INC OUTPTR, $1 ! PUT SPACE!
0300 6185 4306
              893 LD R5, R0_(RB) ! LOAD CONTENT OF 1ST WORD !
0304 0052
               894
                        CALR PHT2
               895
               896 LD R5, R0_+2(R8)
0306 6185 4308
               897
03DA D06D
              898
                       CALR PRIGET
                                       STORE RS, OUTPUT CONOBE!
                             ! & EXPECT INPUT IN R2,R3!
              899
            900 RET C ! RECEIVE Q, RETURN TO DEBUG!
03DC 9E07
                       JR Z, NOIN ! RECEIVE CR, NEXT MEMORY!
03DE E604
              901
03F0 6F82 43D6 902 ID RO (R8), R2 ! REPLACE NEW CONTENTS!
                       LD RO_+2(R8), R3
03F4 6F83 43D8 903
             904
             905 NOIN: INC R8, #4 ! OFFSET+4 NEXT DOUBLE REG !
03F8 A983
             906 INC R9, #4 POINT TO NEXT REG LITERAL!
03EA A993
O3EC 0B09 0E9B 907 CP R9, $LR15+1 ! DONE IF ADDR OF REG LIT!
03F0 E7E8 908 JR C, DOUBL ! REACH SP !
03F2 9E08 909 RET ! RETURN TO DEBUG!
              910
  911 ! HALF DISPLAY BYTE REGISTER, REG NAME 0-7 ONLY !
               912 ! R8 = OFFSET TO REG CONTENT RH6 = 0 FOR L, 2 FOR H!
               913 | R9 = POINTER TO REG LITERAL!
              914
               915 HALF: CP R9, #LR8
                                        ! CHECK IF >REG 7 !
03F4 0B09 0ESC
            916 EOR: JP NC, EROR
03F8 5E0F 014E
               917
03FC B361 FFF8
               918
                        SRL R6, #8
                                        ! R6 = 0 FOR L, 2 FOR H !
              919
                        JR NZ, SAMR !INC ADDR IF LOW ADDR !
        920
0400 EE01
               921
                        INC RB
0402 A980
               922
         923 SAMR:
                             R1, LIT(R6) ! GET 'RL' OR 'RH' !
0404 6161 0EA0 924 LD
0408 DFE5
              925 CALR STRRG1
              926 LD
927 CALR
                              R1, OR9 ! REG NAME !
040A 2191
040C DEE7
                             STRRGI
040E 608D 43D6 926 LDB
                              RL5, RO_(R8) ! GET REG CONTENT!
                     CALR PUTI
        929
0412 DD6F
0414 6900 4386 930
                       INC
                              OUTPTR
                       CALR PGET
                                        ! OUPUT & INPUT IN R2,R3 !
0418 D089 931
              932 RET 933 JR
                              C ! RECEIVE Q, RETURN TO DEBUG!
041A 9E07
                              Z, NINP
041C E602
                              RO_(R8), RL3 ! REPLACE WITH NEW VALUE!
041E 6E8B 43D6
              934
                        LDB
               935 NINF:
                                       ! OFFSET TO NEXT BYTE REG!
               936 INC R8, $1
0422 A980
                                        ! ADVANCE TO NEXT REG!
              937
                        DEC
                             R6,#2
0424 AB51
                                       ! PRINT LOWER BYTE OF SAME!
0426 E6EE
              938
                        JR
                              Z, SAMR
              939
                                        ! REGISTER !
              940
0428 2106 0002
                        LD
                             R6, #2
                                      ! NEXT REG LITERAL !
              941
                            R9, #2
042C A991
                       INC
042E 0B09 0E8C
                                        ! DONE IF ADDR OF REG!
             942
                       CP
                             R9, #LR8
```

```
0432 E7E8
           943 JR C, SAMR ! LITERAL REACH R8 !
           944 RET RETURN TO DEBUG! ON STORE
0434 9F08
          945
            944
            947
                ! COMMAND WITHOUT REGISTER NAME !
               ! PRINT REGISTER 0-15, PC, FCN !
            948
            949
            950 RALL:
                              | PRINT REG 0-15 HEADER |
0436 DDA3
           951 CALR RHED1
           952 CALR PUTRG1 ! AND CONTENTS!
0438 DD93
           953
           954 CALR RHED2 ! PRINT FCW, PC HEADER !
043A DD89
                              ! AND CONTENTS !
           955
956 JP PUTRG2
957
                             ! THEN RETURN !
043C 5E08 08EA!
0440 958 END REGISTER
960
0440
           941 GLOBAL STRRG1 PROCEDURE
            962
           964 1
            965 ! STRRG1 STORE R1 INTO CONOBF.
            966 ! OUTPUT POINTER OF CONORS (OUTPIR) IS INCREMENTED BY 2 !
            967 1
            969
           970 ENTRY
0440 93F4
            971 PUSH @R15, R4
           972 LD R4, OUTPTR ! CONOBF POINTER !
973 LDB @R4, RH1 ! STORE 1ST BYTE!
974 INC R4
975 LDB @R4, RL1 ! STORE 2ND BYTE!
0442 6104 43B6
0446 2E41
0448 A940
044A 2E49
           975
      976
044C 6901 43B6 977 INC OUTPTR, $2
0450 97F4 978 POP R4, @R15
          979 RET
980
0452 9E08
           980
981 END STRRG1
982 ! PAGE EJECT !
0454
            982 ! PAGE EJECT ! 983
          983
0454 984 GLOBAL BRKPNT PROCEDURE
```

```
987 1
               988 | BREAK COMMAND CLEARS PREVIOUS BREAK POINT AND RESET
               989 ! NEW BREAK POINT BY REPLACING THE ORGINAL INSTRUCTION !
               990 | WITH ILLEGAL INSTRUCTION (0ECO) TO CAUSE A TRAP LATER!
               991 | SAVE THE BREAK POINT ADDRESS & ORGINAL INSTRUCTION. |
               992 ! THE BREAK POINT MUST BE SET ON EVEN ADDRESS, AND ALSO!
               993 ! MUST BE IN EXISTING RAM MEMORY.
               994 1
               994
               997 ENTRY
               998
               999
                     ! RESTORE PREVIOUS BREAK POINT INSTRUCTION BEFORE !
                   ! SETTING NEW BREAK POINT !
              1000
              1001
             1002 LD R2, BRKADR
1003 LD R1, INST
0454 6102 43CE
0458 6101 43CC
045C 2F21
             1004 LD 0R2, R1 ! RESTORE LAST BREAK POINT!
045E 4D08 43CE
              1005
                   CLR
                               BRKADE
             1006
                  CLR INST
0462 4D08 43CC
            1007
0466 4D05 4302
                  LD N_/#1 ! CLEAR PREVIOUS BREAK PT!
046A 0001
              1008
                   CALR NXTARG
046C DE93
              1009
                                        ! SKIP OVER COMMAND !
           1010 RET Z ! NO BREAK ADDRESS !
046E 9E06
              1011
              1012 ! GET BREAK ADDRESS & BREAK COUNT, IF BREAK COUNT!
              1013 ! IS NOT GIVEN, DEFAULT TO 1 !
             1014
0470 DEA8
             1015 CALR GETADN ! GET BREAK ADDR IN R3 !
1016 LD R2, R3 ! SAVE BREAK ADDR !
         1016
0472 A132
                                      ! CR AFTER ADDR, NO BREAK !
        1017 JR
                             ZINCT
0474 F605
                                         ! COUNT !
              1018
            1019
                                     ! GET BREAK COUNT !
0476 DEAB 1020 CALR GETADN
0478 8D34
             1021 TEST R3
              1022
                        JR Z, NCT
                                        ! DEFAULT COUNT =1 !
047A E602
047C 6F03 43C2 1023 NCZ:
                       1 D
                              N_1 R3
        1024
              1025 ! SAVE BREAK ADDRESS AND BREAK INSTRUCTION, REPLACE!
            1026 ! WITH ILLEGAL INSTRUCTION TO CAUSE TRAP!
            1027
              1028 NCT:
             1028 NCT:

1029 RES R2,#0 ! MUST BE EVEN ADDRESS !

1030 LD BRKADR,R2 ! SAVE BREAK PT ADDRESS!
0480 A320
0482 6F02 43CE 1030
0486 2121 1031 LD R1, @R2
0488 6F01 43CC 1032 LD INST, R1 ! SAVE ORGINAL INSTRUCTION!
048C 6101 43BE 1033 LD R1, ILEGAL ! REPLACE ILLEGAL INSTRUCT! 0490 2F21 1034 LD @R2, R1
              1035
```

```
0492 2121
            1036
                          R1,0R2 | CHECK TE BREAK PT SET I
0494 4B01 43BE 1037 CP RIJLEGAL ! AT EXISTING RAM
            1038 RET Z
0498 9E06
            1039
049A 4D05 43C2 1040 LD
                           N . #1
049E 0001
04A0 4D08 43CE 1041 CLR BRKADR
04A4 4D08 43CC 1042 CLR INST
04AB 5EC8 014E 1043 JP EROR ! ILLEGAL BREAK PT !
            1044
04AC 1045 END BRKPNT
             1046 ! PAGE EJECT !
             1047
             1048 GLOBAL GOBRK LABEL
     1049 GO LABEL
             1050 SFTINT LABEL
            1051
04AC
            1052 GLOBAL JUMP PROCEDURE
            1053
   1054 [****************************
             1055 1
             1056 ! JUMP COMMAND CHANGES THE USER CURRENT PC AND START !
    1057 ! EXECUTE FROM THAT SPECIFIED PC BY IRET. ( IDENTIFIER,!
            1058 ! FCW, PC ARE ON TOP OF STACK). ALL USER REGISTERS !
            1059 ! ARE RESTORED.
        1060
        1061 | **********************************
            1062
     1063 ENTRY
            1044
            1065 ! GET JUMP ADDRESS, AND SET INTO PC!
          1066
04AC DEB3 1067 CALR NXTARG
04AE 5E06 014E 1068 JP Z, EROR ! JUMP ADDR MISSING!
           1069
04B2 DEC9
           1070
                    CALR GETADN ! GET JUMP ADDR IN R3 !
       1071 RES R3, #0 ! MUST BE EVEN ADDRESS !
04B4 A330
04B6 6F03 43F6 1072 LD
                          PC_, R3 ! CHANGE PC !
          1073
            1074
            1075
                ! CHECK JUMP ADDRESS MATCH BREAK POINT ADDRESS !
  1076
          1077 ! IF PC DDES NOT MATCH BREAK ADDRESS, RESTORE ALL !
                 ! ALL REGISTERS, AND RETURN TO USER PROGRAM THROUGH!
            1078
            1079
                  ! DATA ON STACK !
         1080
04BA 6103 43F6 1081 GO: LD R3, PC_
04BE 4B03 43CE 1082 CP R3, BRKADR ! CHECK IF NECESSARY TO !
04C2 E604 10B3 JR Z, GOBRK ! REPLACE INSTRUCTION & !
         1084
                                  ! CAUSE CTC INTERRUPT !
04C4 DF4B 1085 CALR RESTOR ! NO NEED, JUST RESTORE REG! 04C6 610F 43F4 1086 LD R15, R15_
```

```
04CA 7800
                     1087 IRET
                                                 TR TREATER EL
                        1088
                              1089
                            1092 ! IF PC MATCH BREAK ADDR., RESTORE ORGINAL INSTRUCTION!
                              1093 ! AT BREAK ADDRESS, AND CAUSE INTERRUPT HAPPEN
                              1094 ! AFTER EXCUTION OF THAT INSTRUCTION
                              1095 1
                              1097
                             1098
O4CC 2102 0574' 1099 GOBRK: LD R2, #BRKINT ! BREAK SERVICE ROUTINE!
                             1100
04D0 6103 43CE 1101 LD R3, BRKADR
                                           LD R1, INST
04D4 6101 43CC 1102
04D8 2F31 1103 LD 0R3, R1 | RESTORE INSTRUCTION!
                     THE 1104 HOLD BAR IN LITE WAS BUILDING TO BE THE THE THE PARTY OF THE 
                             1105
                             1107 1
                             1108 ! SET INTERRUPT SERVICE ROUTINE AT PROGRAM STATUS AREA
                            1109 ! RESTORE USER REGISTERS, PROGRAM CTC3 TO COUNTER MODE!
                             1110 ! AND INTERRUPT AFTER EXECUTE THE ORGINIAL INSTRUCTION !
                             1111 ! AT BREAK POINT. SAVE USER FOR TO RESTORE LATER.
                             1112 ! TURN ON VECTOR INTERRUPT TO CAUSE CTC TO INTERRUPT.
                             1113 | AND CONTINUE EXECUTING BY IRET.
              1114 1
                             1116
                    1117 SFTINT:
                                                            VI ! TURN OFF INTERRUPT FOR!
04DA 7C01
                             1118 DI VI
                             1119
04DC 6F02 442A 1120 LD PSCTC, R2
                                                                                        ! PUT INTERRUPT ADDRESS !
                             1121
                                                                                         ! IN FROG. STATUS AREA!
                            ! USE VECTOR 6 !
                            1123
                             1124
04E0 DF59
                                              CALR RESTOR
                                                                                        ! RESTORE REG, PC, FCW !
                             1125
04E2 C900 1126
04E4 3A96 FF11 1127
                                                 LDB RLi, #00
                                                  GUTB CTCO, RL1
                                                                                        ! INTERRUPT VECTOR !
                             1128
                                                                                         ! CTC3 SEND VECTOR 6!
04E8 C9C7
                             1129
                                          LDB RL1, #CNTMOD
04EA 3A96 FF17 1130 OUTB CTC3, RL1 ! COUNTER MODE!
                             1131
                             1132
                                       ! SET CTC TIME CONSTANT TO 4 FOR 4 STACK OPERATION!
                            1133
04EE 5101 43C0
                            1134
                                                 LD R1 , TIMENT
                                                  OUTB CTC3, RL1 ! TIME CONSTANT !
04F2 3A96 FF17
                            1135
                            1136
04F6 610F 43F4
                                               LD R15, R15_ 7334 043 8841
                            1137
                             1138
```

```
04FA 31F1 0002 1139 LD R1, R15(#2) ! 1ST STACK ! 04FE 6F01 43B2 1140 LD USFCH, R1 ! CHANGE FC --> EI VI ! 0502 0501 1000 1141 DR R1, #VINT
0506 33F1 0002 1142 LD R15(#2), R1 ! 2ND STACK!
050A 6101 43D8 1143 LD R1, R1_
050E 7B00 1144 IRET ! 3RD & 4TH STACK!
         1145 THE TOTAL THE T
             1146 END JUMP
0510
             1147 | PAGE EJECT |
            1148
0510
             1149 GLOBAL NEXT PROCEDURE
  1152 1
              1153 ! NEXT STEP THROUGH USER PROGRAM INSTRUCTIONS BY
             1154 | INSTRUCTIONS
    1155 ! IF THE CURRENT PC MATCH THE BREAK POINT ADDRESS,
             1156 ! THE NEXT COMMAND WILL CLEAR THE BREAK POINT.
             1157 1
         1159
   1160 ENTRY
  1161 ! GET NEXT COUNT, DEFAULT TO ONE, PRINT HEADER!
         1162
0510 DEE5 1163 CALR NXTARG
0512 E603 1164 JR Z, D1 ! NO COUNT!
1165
0514 DEFA 1166 CALR GETADN
                                      ! GET COUNT IN R3 !
0516 8D34 1167 TEST R3
0520 DE18 1171 CALR RHED1
0522 DDFD 1172 CALR RHED2
  1173
           1174 ! IF PC MATCH BREAK ADDRESS, RESTORE ORGINAL !
             1175 ! INSTRUCTION AND CLEAR THE BREAK POINT!
     117ò
              1177 NXT1:
0524 2102 053E' 117B LD R2, $TRCINT ! INT. SERVICE ROUTINE!
1179
0528 6103 43CE 1180 LD R3, BRKADR
052C 4803 43F6 1181 CP R3, PC_
                                           ! FOR NEXT !
0530 EED4 1182 JR NZ, SFTINT
                                           ! THE NEXT INSTRUCTION NOT!
             1183
                                           ! AT BREAK POINT !
0532 6103 43CE 1184 LD R3,BRKADR
0536 6101 43CC 1185 LD R1, INST
053A 2F31 1186 LD @R3, R1
                                           ! PUT BACK ORGINIAL INST.!
053C E8CE 1187
                      JR SFTINT
             1188
053E
            1189 END NEXT
```

```
1190 ! PAGE EJECT !
              1191
053E
             1192 GLOBAL TROINT PROCEDURE
             1193
             1195 !
             1196 | CTC INTERRUPT FOR NEXT
             1197 ! INTERRUPT SERVICE ROUTINE FOR NEXT INSTRUCTION
             1200
             1201 ENTRY
            1202 LD R15_/ R15
1203 LD R0_, R0
1204 CALR SAVREG
053E 6F0F 43F4
0542 6F00 43D6
            1203
                                        ! SAVE RO,R15 FIRST !
0546 DF9D
                                        ! SAVE STACK INFORMAT.!
             1205
                                        ! AND REGISTER 1-14 !
            1206
1207 CALR RESCTC
1208 LD R2,BRKADR
1209 LD R1,ILEGAL
1210 LD @R2,R1
0548 DFA8
                                        ! RESET CTC !
054A 6102 43CE
054E 6101 43BE
                                         ! PUT BACK ILLEGAL INST.!
                                       ! IN CASE GOT CLEAR !
0552 2F21
            1211
1212 BIT
1213 JR
1214 RES
1215 NFIX:
0554 670C 43D2
                             USFCW, #BVINT
                                       ! FIX VI IN FCH !
0558 EE02
                             N7, NFTY
055A 630C 43F8
                             SAVECH, #BVINT
             1216 CALR
1217 CALR
055E DE26
                             PHTRGI
0560 DE3C
                                       ! PUT OUT REG CONTENT!
                             PUTRG2
0562 6B00 43C2 1218
                    DEC
                             N_
                      JR
                            NZ, NXT1 ! CONTINUE ON !
0566 EEDE
             1219
             1220
            1221 LD N_, $1
                                       ! SET COUNT TO 1 AGAIN!
0568 4D05 43C2
056C 0001
056E 7C05
                    EI VI
JP DEBUG
             1222
                                     ! ALLOW INTERRUPT!
Q570 5E08 00EC' 1223
             1224
             1225 END TRCINT
             1226 ! PAGE EJECT !
             1227
             1228
0574
             1229 GLOBAL BRKINT PROCEDURE
             1230
             1232 !
             1233 ! CTC INTERRUPT FOR BREAK
             1234 ! INTERRUPT SERVICE ROUTINE FOR EXCUTING ORGINAL
             1235 ! INSTRUCTION AT BREAK POINT, PUT BACK ILLEGAL
             1236 ! INSTRUCTION AND RESTORE USER FCW
             1237 !
             1239
             1240 ENTRY
```

```
@R15, R2
            1241
                       PHSH
0574 93F2
              1242
0576 93F1
                        PUSH
                               PR15, R1
              1243
                               RESCTC
                   CALR
0578 DECO
              1244
              1245
057A 6102 43CE
              1246
                         1 D
                               R2, BRKADR
                        LD R1, ILEGAL
057E 6101 43BE
              1247
              1248
                     LD
                               @R2, R1 ! PUT BACK ILLEGAL INST!
0582 2F21
              1249
              1250
                        BIT
0584 6700
                               USECH, #BVINT
        43D2
                         JR
              1251
0588 FF05
                               NZ, NOFTX
                                            I DEBUG SET VI ALREADY !
058A 31F2 0006
              1252
1253
                        LD
                               R2, R15(#6)
                                          STACK ORDER:R2,R1,!
              1254 RES R2, #BVINT
1255 LB R15 (#6), R2
                                            ! IDENT, FCM, PC !
                                           ! RESET VI IN FCM !
058F A32C
0590 33F2 0006
              1256
           1257 NOFIX: POP R1, @R15
1258 POP R2, @R15
0594 97F1
0596 97F2
                        IRET
0598 7B00
              1259
              1260
              1261 END BRKINT
059A
              1262 ! PAGE EJECT !
              1263
              1264 GLOBAL NMINT PROCEDURE
059A
              1265
              1267 !
              1268 ! NMINT SERVICE ROUTINE RESPONSE TO THE NMI SWITCH. !
              1269 ! IF SYSTEM IS IN TRANSPARENT MODE, IT WILL SEND AN !
              1270 ! ESCAPE KEY TO MCZ, AND RESET THE STACK.
              1271 ! IF SYSTEM IS RUNNING USER PROGRAM, IT SAVE ALL USER !
              1272 ! REGISTERS, PC, FCW.
              1273 !
              1275
              1276 ENTRY
              1277 BIT CFLAGS, #TRNMDE ! CHECK IF IN TRANSPARENT!
059A 6700 43D4
                        JR NZ,QTRAN ! YES, SEND ESCAPE !
059E EE06
              1278
              1279
                        LD
                                            ! SAVE USER REGISTER...!
05A0 6F0F 43F4
                               R15_,R15
              1280
                        LD
05A4 6F00 43D6
              1281
                               RO RO
05A8 DFCE
              1282
                         CALR
                               SAVREG
              1283
                        JR
                               PNMI
05AA E808
               1284
              1285 QTRAN: RES
                               CFLAGS, #TRNMDE ! RESET TRANSPARENT BIT !
05AC 6300 43D4
                        LD
05B0 210F 40A0
              1286
                               R15,#STACK
                                          ! RESET STACK POINTER !
05B4 6F0F 43F4
              1287
                         LD
                               R15_/R15
05B8 C81B
               1288
                         LDB
                               RLO, TESCHAR ! SEND ESC KEY TO MEX!
05BA DF17
                         CALR
                                MCZSND
               1289
               1290
               1291 PNMI: EI
05BC 7C05
                                VI
                               CRLF
OSBE DF10
              1292
                         CALR
```

```
05C0 2102 0EAE 1293 LD R2, #NMIMSG ! PRINT NMI MESSAGE ! 05C4 DE89 1294 CALR PUTMSG 05C6 5E08 00EC' 1295 JP DEBUG
             1296
              1297 END NMINT
              1298 ! PAGE EJECT !
 05CA
              1300 GLOBAL BREAK PROCEDURE
              1303 !
             1304 ! BREAK CAUSE BY ILLEGAL INSTRUCTION, SAVE USER REGISTER!
              1305 ! PC & FCW. CHECK BREAK COUNT, CONTINUE IF COUNT NOT 1 !
              1308
              1309 FNTRY
05CA 6F0F 43F4 1310 LD R15_, R15
05CE 6F00 43D6 1311 LD R0_,R0
05D2 DFE3 1312 CALR SAVREG
                                      ! SAVR RO, IEDNT, FCW, PC!
                                      ! AND REGISTER !
05D4 6B01 43F6 1314 DEC PC_, #2 ! BACK UP TO BREAK ADDR !
              1315
              1316
05D8 6B00 43C2 1317 DEC N_, #1 ! DEC BREAK COUNT!
05DC 5E0E 04CC 1318 JP NZ, GOBRK ! CONTINUE IF NOT 1!
05E0 4D05 43C2 1319 LD N_, $1 ! SET BREAK COUNT=1 !
05E4 0001
05E6 7C05 1321 EI VI ! ALLOW KB
05E8 2102 0EA4 1322 LD R2, #BRKMSG
1323
                                         ! ALLOW KB INTERRUPT !
05EC DE9D 1324 CALR PUTMSG 05EE 6105 43F6 1325 LD R5, PC_ ! PUT OUT MESSAGE WITH! 05F2 DE61 1326 CALR PUT2 ! BREAK ADDRESS!
05F4 DE40 1327 CALR PUTPTR
05F6 5E08 00EC' 1328 JP DEBUG
           1329
              1330 END BREAK
        1331 ! PAGE EJECT !
              1332
05FA 1333 GLOBAL RESCTC PROCEDURE
              1334
              1337 ! RESET CTC BY SIMULATE Z80 RETI INSTRUCTION AND
              1338 ! SEND DATA 03 TO CTC3
              1339 1
             1341 ENTRY
05FA C9ED 1342 LDB RL1, #XED ! SIMULATE Z80 RETI ! 05FC 3A96 FF29 1343 OUTB RETI, RL1
```

```
0600 C94D 1344 LDB RL1, #%4D
0602 3A96 FF29 1345 OUTB RETI, RL1
0606 C903 1346 LDB RL1, #%3
0608 3A96 FF17 1347 OUTB CTC3, RL1 ! RESET CTC3 !
060C 9E08 1348 RET
1349
060E
            1350 END RESCTC
            1351
            1352
            1353 GLOBAL SAVREG PROCEDURE
            1754
            1356 1
            1357 ! SAVREG SAVE USER FCW AND PC FROM STACK, ALSO SAVE
            1358 ! ALL USER REGISTER CONTENTS REG 1-14
            1361
            1362 ENTRY
060E 6F0F 43D0 1363 LD RETSP, R15 ! SAVE RETURN ADDRESS!
0612 31F0 0004 1364 LD RO, R15(#4) ! EXTRA RETURN ADDR, IDENT.!
0616 6F00 43F8 1365 LD SAVFCW, RO ! BEFORE USER FCW !
061A 31F0 0006 1366 LD RO, R15(#6) ! USER PC ! 
061E 6F00 43F6 1367 LD PC_, RO
            1368
           1369 LD R15, #R1_
0622 210F 43D8
0626 1CF9 010D 1370 LDM @R15, R1, #14 ! SAVE REG 1-14 !
062A 610F 43D0 1371 LD R15, RETSP 062E 9E08 1372 RET 1373
                    RET
            1374 END SAVREG
0630
            1375
            1376
        1377 GLOBAL RESTOR PROCEDURE
            1378
            1380 1
            1381 ! RESTOR RESTORE ALL USER REGISTER, PC, & FCW
            1384 ENTRY
0630 6F0F 43D0 1385 LD RETSP, R15
0634 610F 43F4 1386 LD R15, R15_
                        RETSP, R15 ! SAVE RETURN ADDRESS !
0638 6101 43F8 1387 LD R1, SAVFCW
063C 33F1 0002 1388 LD R15(#2), R1
0640 6101 43F6 1389 LD R1, PC_
0644 33F1 0004 1390 LD R15(#4), R1
            1391
0648 210F 43D6
            1392 LD R15, #R0_
064C 1CF1 000E 1393 LDM RO, @R15, #15 ! RESTORE REG 0-14 !
0650 610F 43DO 1394 LD R15, RETSP
0654 9E08
            1395
                     RET
```

```
1396
0656 1397 END RESTOR
1398 ! PAGE EJECT !
            1399
            1400 1
                    UTILITY SUBROUTINES
            1401
0656
            1402 GLOBAL GETNUM PROCEDURE
  1403
 1405 1
            1406 ! GETNUM GET 1 LINE OF INPUT AND PACK 8 ASCII CHAR INTO
            1407 ! DOUBLE WORD HEX VALUE. IF RECEIVE LESS THAN 8 CHAR
            1408 ! FILL WITH LEADING HEX ZERO
           1410 ! OUTPUT: R2, R3 = EIGHT 4-BIT HEX VALUE
           1411 1
           1412 1
                    RET C FOR RECEIVING 'D'
   1413 ! RET NC, Z, IF CARRIAGE RETURN ONLY
   1414 ! RET NC,NZ, IF CHARACTERS FOLLOMED BY SPACE
      1415 JUMP TO ERROR IF RECEIVE NON-ASCII CHAR
     1416 1
  1417 ! *** RO, R1, R2, R3 ARE LOST ***
           1418 1
           1420
          1421 ENTRY
0656 C83C
           1422 LDB
                         RLO, #1(1
         1423 CALR TYWR
0658 DF72
      1424 CALR GET
065A DF4E
                                 ! GET LINE IN CONIBF !
           1425
065C 8D28 1426 CLR R2
                                ! R2, R3=0 !
065E 8D38 1427 CLR R3
0660 DF88
                  CALR SKIPSP
       1428
                                 ! SKIP LEADING BLANK !
           1429 RET Z ! RET NC/Z FOR RECEIVE CR!
0662 9E06
           1430
0664 0A08 5151 1431 CPB RLO , #'Q' ! CHECK IF QUIT
       1432 JR NZ, XMORE ! NO
0668 EE02
           1433
1434
066A 8D81
                  SETFLG C
066C 9E08
                  RET
                                 ! RETURN C FOR 'Q'
1436
           1437 XMORE: CALR CONVERT
                                ! CONVERT 8 BIT ASCII CHAR!
066E DFBB
        1438 ! TO 4 BIT HEX VALUE !
0670 E75F
          1439 JR C, ERRE! BAD CHAR!
                  RLDB RLO, RL3 ! SHIFT HEX VALUE INTO !
0672 BEBB
           1440
0674 BE38 1441
                  RLDB RLO, RH3 ! RIGHT MOST LONG WORD IN!
        1442 RLDB RLO, RL2
1443 RLDB RLO, RH2
                                ! R2, R3
0676 BEAR
0678 BE28
           1444
                  CALR GCHAR ! NEXT CHARACTER, RET NC & !
067A DF68
           1445
         1446
                             ! Z FOR CR !
                  JR Z, RNZ
067C E604
          1447
```

```
1448
                                      I NC TS RESULT FROM GCHAR I
             1449
                                    ! CP INSTRUCTION !
                    CPB RLO, #' '
067F 0A0B 2020
                                    ! IF HIT ' ', END OF INPUT!
             1450
0682 FFF5
             1451
                      JR
                           N7, XMORE
             1452
             1452
1453
0684 8083
                       RESFLG C
             1454
             1455 RNZ: RESFLG Z
0686 8043
                                ! RETURN NZ,NC FOR INPUT !
0688 9E08
            1456 RET | FOLLOW BY SPACE OR CR |
             1457
048A
            1458 END GETNUM
            1459 ! PAGE EJECT !
             1460
0684
             1461 GLOBAL TTY PROCEDURE
            1462
             1464 1
          1465 ! RECEIVE AND ECHO INPUT LINE, STORE LINE IN RUFFER
   1466 ! ADDRESS BY R2, SIZE OF BUFFER IS SPECIFIED BY R1.
     1467 ! TERMINATE LINE WHEN RECEIVE CARRIAGE RETURN
             1468 ! THIS ROUITINE ALSO HANDLE DELETE CHARACTER AND
             1469 ! LINE DELETE CHARACTER, ALL CHARACTER ARE STORED IN
             1470 ! UPPER CASE
            1471 !
             1472 ! INPUT: R1 = SIZE OF BUFFER IN BYTE
             1473 I
                         R2 = BUFFER ADDRESS
             1474 1
             1475 ! OUTPUT: R1 = # OF CHARACTERS RECEIVED
    1476 !
                        INPUT LINE STORED IN BUFFER ADDRESSED BY R2
             1477 !
             1478 !
                         RETURN Z IF BUFFER SIZE EXCEED LIMIT
             1479 1
                         RETURN NZ FOR RECEIVING INPUT LINE WITH CR !
   1480 !
 1481 ! *** RO, R1, R2 ARE LOST ***
             1482 !
         1484
             1485 ENTRY
            1486
068A 6F02 43B0
                      LD
            1487
                            CBUFADR, R2 ! SAVE BUFFER ADDR & SIZE!
            1488
068E 6F01 43B2
                      LD
                            CBUFSIZ, R1
             1489
0692 6102 43B0
            1490 BEGN: | I D
                            R2, CBUFADR ! BUF ADDRESS!
0696 8D18
             1491
                      CLR
                            R1 ! COUNT !
             1492
0698 DFD2
            1493 RDLOP: CALR TTYRD ! GET CHAR & ECHO!
             1494
             1495
                  ! CONVERT ALL LOWER CASE ALPHA TO UPPER CASE !
             1496
069A 0A08 6161 1497
                      CPB RLO, #%61
069E E704
            1498 JR
                            C, NOCHK ! NOT LOWER CASE !
06A0 0A08 7B7B 1499
                      CPB RLO, #%7B
```

```
06A4 FF01
         1500 JR NC, NOCHK
0686 A285
            1501 RESB
                            RLO, #LONCSE ! CONVERT TO HPPER CASE !
             1502
             1503 NOCHK:
06AB 2E2B
                            @R2, RLO
        1504 LDR
06AA 4A08 43B8
            1505
                      CPB
                            RLO, CHRDEL! CHECK FOR DELETE CHAR!
OGAF FEOD
                 JR
             1506
                            NZ, CKMRE
            1507
06B0 AB20
             1508
                      DEC
                            R2, #1 ! RECEIVE DEL, BACK UP 1 !
06B2 AB10
             1509
                      DEC
06B4 4B02 43B0 1510 CP
                            R2, CBUFADR ! CK TOO MUCH BACK SPACE !
            1511
                      JR
                            C, STARTS ! BACKUP TOO MUCH !
06B8 E705
                            RLO, #1 1
            1512 LDB
06BA C820
OARC BEA4
            1513
                      CALR
                            TYWR
            1514
                      LDB
                            RLO, #BACKSP ! ERASE LAST CHAR !
06BE C808
                    CALR
OACO DEAA
           1515
                            TYMR
O&C2 EBEA
            1516
                     JR
                            RDI OP
            1517
                            RLO, #'()
            1518 STARTS: LDB
06C4 C83C
                            TYMR ! PREVENT BACKUP TO WRAP!
            1519 CALR
OACA DEA9
                     JR BEGN ! AROUND !
06C8 EBE4
            1520
            1521
06CA 4A08 43B9 1522 CKMRE: CPB RLO, LINDEL! CK FOR LINE DELETE!
06CE E609
            1523
                 JR
                         Z, DELIN ! YES, START AGAIN!
                         R2, $1 ! ADVANCE BUF ADDR & COUNT!
            1524 INC
06D0 A920
06D2 A910 1525 INC R1
06D4 0A08 ODOD 1526 CPB RLO, #ASCICR ! CK FOR CR !
            1527 JR Z, FINLIN! YES, TERMINATE WITH LF!
06D8 E60A
                 CP
           1528
                            R1, CBUFSIZ ! CHECK IF EXCEED BUF SIZE!
06DA 4B01 43B2
            1529 JR NZ, RDLOP ! NO, GET NEXT!
OADE EEDC
        1530 RET Z
                                    ! ERROR, TOO MANY CHAR !
06E0 9E06
06E2 C85E
          1532 DELIN: LDB
                            RLO, #%5E
                                    ! CHAR FOR DELETE LINE !
06E4 DFB8
            1533 CALR TYNR
                 CALR
LDB
06E6 DFA4
            1534
                            CRLF ! FORGET THIS LINE, START !
         1535
                            RLO, #1 1
06E8 C820
                            TYWR
       1536 CALR
OSEA DEBB
                            BEGN
OSEC EBD2
                  JR
            1537
                                    ! INPUT ON NEXT LINE !
            1538
          1539 FINLIN: LDB
                            RLO, #ASCILF
OSEE CBOA
                            TYHR ! OUPUT LF !
            1540 CALR
OAFO DEBE
            1541
            1542
         1543 RESFLG Z ! RETURN NZ FOR OK !
06F2 8D43
            1544
                    RET
06F4 9E08
            1545
            1546 END TTY
06F6
```

```
1547
             1548
OAFA
             1549 GLOBAL TTYRD PROCEDURE
             1550
             1551 | ****************************
             1553 ! TTYRD GET SINGLE CHARACTER FROM INTERRUPT INPUT BUFFER!
             1554 ! AND ALSO ECHO BACK TO TTY. IF BUFFER EMPTY, HATT UNTIL!
             1555 ! IT RECEIVE CHARACTER
             1557 ! OUTPUT: RLO = CHAR & RETURN Z IF CHAR IS CR
             1560
            1561 ENTRY
OAFA DECD
            1562 CALR TYIN
06FB E83E
             1563
                      JR TYNR
06FA
            1564 FND TTYRD
             1565 ! PAGE EJECT !
            1566
06FA
             1567 GLOBAL CONVERT PROCEDURE
       1571 ! CONVERT CHANGE 8 BIT ASCII CHAR. TO CORRESPONDING
             1572 ! 4 BIT HEX VALUE, IF CHARACTER OTHER THAN 0-9, A-F,
             1573 ! JUMP DIRECT TO ERROR , FIX STACK, PRINT ? & RE-ENTER !
    1574 ! DEBUG. (r.e. 39-->9 41-->A)
   1575 !
            1576 ! INPUT: RLO = 8 BIT ASCII CHARACTER
            1577 !
             1578 ! OUTPUT: RLO = 4 BIT HEX VALUE NITH LEADING ZERO
            1579 !
            1580 !
                        RETURN C IF CHAR IS NON-ASCII
            1581 ! RETURN NC IF OK
            1584
            1585 ENTRY
04FA 0A08 3030
            1586
                      CPB RLO, #'0' ! NO CHAR ( 30H
                                                    CASE PERON
                         C ! BAD CHAR
06FE 9E07
            1587 RET
                           RLO, #'9'+1 ! CK IF DIGIT
0700 0A08 3A3A
           1588
                      CPB
0704 F708
                      JR
            1589
                           C, NOSUB
0706 0A08 4141 1590 CPB
                            RLO, #'A' ! DON'T ALLOW 40H
            1591
070A 9E07
                      RET
                            C
070C 0A08 4747
            1592
                      CPB
                           RLO, #'F'+1 ! NO ALPHA > F
0710 EF06
            1593
                      JR
                           NC, RETC
0712 0208 0707
            1594
                      SUBB
                           RLO, #7
                                    ! FIX UP IF ALPHA
                           RLO, #XOF ! ONLY NEED 4 BITS
0716 0608 OFOF
           1595 NOSUB:
                      ANDB
071A 8D83
            1596
                      RESFLG C
071C 9E08
            1597
                      RET
            1598
```

```
1601
           1602 END CONVERT
    1603
          1604
0722
            1605 GLOBAL GETADN PROCEDURE
            1408 1
            1609 | GETADN CONVERT NEXT ARGUMENT IN CONIBE BUFFER &
             1610 ! CONVERT TO A HEX ADDRESS, ONLY TAKE LAST 4 CHARACTERS !
            1611 ! 1ST CHARACTER OF THE ARGUMENT IS ALREADY IN RLO
            1612 1
       1613 ! INPUT: RLO - 1ST CHAR OF THE ARGUMENT
            1614! HL - POINTER TO 2ND CHAR OF THE ARGUMENT
            1615 1
            1616 1
       1617 ! OUTPUT: R3 = HEX VALUE NUMBER
            1618 !
            1619! RET Z.C IF ONLY CARRIAGE RETURN
            1620 ! RET Z,NC IF ARGUMENT FOLLOWED BY CR
1621 ! RET NZ,NC IF ARGUMENT FOLLOWED BY SPACE
                      JR TO ERROR, IF RECEIVE CHARACTER OTHER THAN !
             1622 !
            1623 ! 0-9/A-F/CR
            1624 1
            1625 ! *** RO IS LOST ***
            1626 !
             1627 | **********************************
            1628
            1629 ENTRY
1634 RET ! CR ONLY !
072C 9E08
            1635
          1636 NCR:
                           CONVERT ! RETURN 4 BIT HEX VALUE !
072E D01B
            1637 CALR
0730 5E07 014E 1638 ERRE: JP C, EROR ! BAD CHAR !
                           RLO, RL3 ! SHIFT VALUE IN RIGHT !
0734 BEB8 1639 RLDB
                     RLDB
                           RLO, RH3
                                   ! MOST POSITION
0736 BE38
            1640
            1641
                                    ! NEXT CHAR IN CONIBF !
0738 DFC7
            1642 CALR
                                    ! (CP RLO, #ASCICR) WILL !
            1643
                                    ! RETURN NC !
            1644
            1645
                           Z ! RETURN Z, NC FOR RECEIVE!
           1646 RET
073A 9E06
                                   ! CR AFTER ARGUMENT !
            1647
            1648
                    CPB
                           RLO, #' ' ! CHECK IF SPACE !
073C 0A08 2020 1649
            1650
                           NZ, NCR ! NO, CONVERT NEXT CHAR!
0740 EEF6
                    JR
```

```
1651 CALR SKIPSP ! SKIP OVER BLANK !
0742 DEF9
          1652
                                    I AND GET NEXT NON-BLANK
             1653
                                    ! CHAR IN RLO !
           1654 RESFLG C
1655 RET
0744 8D83
0746 9F0B
                                      ! RETURN NZ,NC FOR SPACE !
            1656
                                      ! AFTER ARGUMENT !
             1657
0748
             1658 END GETADN
             1659
             1660
0748
             1661 GLOBAL NXTARG PROCEDURE
             1664 1
             1665 ! NXTARG SKIP OVER CURRENT ARGUMENNT AND THEN BLANK
             1666 ! TO FIRST NON-BLANK CHARACTER
             1668 ! OUTPUT: RLO = NON-BLANK CHARACTER OR CR
             1669 ! RETURN Z IF CHARACTER IS CARRIAGE RETURN
             1670 1
             1672
             1673 ENTRY
16/3 ENTRY

0748 DFCF 1674 CALR GCHAR

074A 9E06 1675 RET Z ! RET FOR HIT CR !

074C 0A08 2020 1676 CPB RL0, *' ! SKIP CURRENT ARGUMENT !

0750 EEFB 1677 JR NZ, NXTARG
             1678
            1679 ! FALL THROUGH TO ROUTINE SKIPSP
            1681 END NXTARG
            1682
             1683
0752
             1684 GLOBAL SKIPSP PROCEDURE
             1685
             1688 ! SKIPSP OVER BLANK TO FIRST NON-BLANK CHARACTER
             1689 1
             1690 ! OUTPUT: RLO = CHARACTER
             1691! RETRUN Z IF CHARACTER IS CARRIAGE RETURN
             1692 !
             1694
   1695 ENTRY
            1696
0752 BFD4
0752 BFD4 1697 CALR GCHAR
0754 9E06 1698 RET Z
                                   ! RECEIVE CR !
0756 0A08 2020 1699 CPB RLO, #' '
            1700 JR Z, SKIPSP
1701 RET
075A E6FB 1700
075C 9E08
      1702
```

```
075E
              1703 END SKIPSP
             1704
               1705
075F
               1706 GLOBAL TYIN PROCEDURE
               1707
               1700 | BERRESERVERSERVERSERVERSERVERSERVERSERVERSERVERSERVERSERVERSER
              1709 1
               1710 ! GET CHAR FROM INTERRUPT INPUT RINGRE, NO ECHO ON TTY !
        1711 !
           1712 ! OUTPUT: RLO = CHAR
               1713 1
               1714 ! *** RO IS LOST ***
       1715 !
               1717
               1718 ENTRY
075E 6100 43C6 1719 LD RO, TTYGET ! IF TTYGET=TTYPUT, THEN!
0762 4800 43C4 1720 CP RO, TTYPUT ! BUFFER EMPTY!
0766 E6FB 1721
1722
0768 93F2 1723 PUSH 0R15, R2
076A DFC7 1724 CALR GBFADR ! GENERATE RINGBF ADDR!
076C 6F00 43C6 1725 LD TTYGET, RO ! UPDATE RINGBF GET CHAR!
1726 ! POINTER!
0770 2028 1727 LDB RLO, 0R2 ! GET THE CHARACTER!
0772 97F2 1728 POP R2, 0R15
            1731 END TYIN
             1732
              1733
0776
              1734 GLOBAL TYWR PROCEDURE
              1735
           1737 1
               1738 ! TYWR CHECK IF XOFF CHARACTER IS RECEIVED BEFORE ECHO!
               1739 ! CHARACTER IN RLO TO THE TERMINAL
               1740 !
              1741 ! OUTPUT: RLO = SAME CHARACTER
              1742 ! RETURN Z IF OUTPUT CHAR IS CR
              1743 !
               1745
              1746 ENTRY
0776 6702 4304
              1747 BIT CFLAGS, #OUTHLD ! DON'T OUTPUT IF RECEIVE!
077A EEFD
              1748
                        JR NZ, TYNR ! XOFF CHAR, WAIT.....!
              1749
077C 3A04 FF1F
             1750
                        INB RHO, SIOCB
0780 A602
                        BITB RHO, #TXRDY
              1751
                    JR
0782 E6F9
              1752
                              Z, TYWR
0784 3A86 FF1B 1753
                       OUTB SIODB, RLO
                   CPB RLO, #ASCICR
0788 0A08 0D0D 1754
```

```
1755 RET 1291142 019 LOST
078C 9F08
078E
            1756 END TYNR
            1757
            1758
            1759 GLOBAL MCZSND PROCEDURE
            1740
            1762 1
            1763 ! MCZSND OUPUT 8-BIT DATA IN RLO TO STO CHANNEL A
            1764 ! DATA PORT. (SEND CHARACTER IN RLO TO MCZ)
            1765 !
            1766 | INPUT: RIO = 8 BIT DATA
            1767 !
                     RETURN Z IF CHARACTER IS CARRIAGE RETURN
            1768 1
            1769 | *******************************
            1770 ENTRY
078E 3A04 FF1D 1771 INB RHO, SIOCA
0792 A602 1772 BITB RHO, $TXRDY ! CHECK IF TRANSMIT READY!
           1773 JR Z, MCZSND
0794 E6FC
0796 3A86 FF19 1774 OUTB SIODA, RLO ! OUTPUT DATA!
079A 0A08 ODOD 1775 CPB RLO, #ASCICR
079E 9E08 1776 RET
         1777 END MCZSND
07A0
            1778
           1779
            1780 GLOBAL CRLF PROCEDURE
07A0
            1781
            1784 ! CRLF OUTPUT CARRIAGE RETURN AND LINE FEED
            1786 ! *** RO IS DESTROYED ****
            1787 !
           1790 ENTRY
07A0 C80D 1791 LDB RLO,#ASCICR
       1792 CALR TYWR
1793 LDB RLO, #ASCILF
1794 CALR TYWR
07A2 D017
07A4 C80A
                  CALR TYWR
07A6 D019
07A8 5E08 0962' 1795 JP
                         NULL
                                 ! PRINT NULL CHAR IF ANY !
07AC 1796 END CRLF
           1797
           1798
07AC 1799 GLOBAL GCHAR PROCEDURE
  1800
```

```
1803 | FETCH A CHARACTER FROM CONTRE THEN INCREAMENT RUFFER
             1804 ! POINTER (INPTR) BY 1
              1805 1
   1806 ! OUTPUT: RLO = CHARACTER
             1807 1
              1808 ! RETURN Z IF RECEIVE CR
             1811
              1812 ENTRY
07AC 93F2 1813 PUSH @R15, R2
07AE 6102 43B4 1814 LD R2, INPTR
07B2 2028 1815 LDB RL0, @R2 ! FETCH CHAR !
07B4 6900 43B4 1816 INC INPTR, $1 ! ADVANCE CONIBF POINTER!
07B8 0A08 0D0D 1817 CPB KLU7 VIII
07BC 97F2 1818 POP R2, QR15
07B8 0A08 0D0D 1817 CPB RLO, #ASCICR
       1820 END GCHAR
0700
         1821
             1822
0700
              1823 GLOBAL GET PROCEDURE
              1827 ! FETCH SINGLE INPUT LINE IN CONIBE, MAXIMUM 80 CHARACTER!
             1828 ! & LINE IS TERMINATE BY CR. ALL CHARACTERS RECEIVED !
             1829 I HAVE BEEN CONVERTED TO UPPER CASE AND ECHOED TO ITY
             1830 !
              1831 ! OUTPUT: RLO = 1ST CHAR IN CONIBF
              1832! RETURN Z IF CHARACTER IN RLO IS CR
                        INPTR = CONIBF PTR TO FETCH NEXT INPUT CHAR
                        (STILL POINT TO 1ST CHAR)
              1834 !
              1835 !
              1836 ! *** RO, R1, R2 ARE LOST ***
             1837 !
              1839
             1840 ENTRY
07C0 2102 40B0 1841 LD R2, #CONIBF
07C4 2101 0080 1842 LD R1, #CISIZE
             1843 LD INPTR, R2 ! CONIBF INPUT POINTER!
07C8 6F02 43B4
             1844 CALR TTY ! INPUT LINE IN CONIBF!
O7CC DOA2
07CE 5E06 014E 1845 JP Z, EROR ! EROR, TOO MANY CHAR!
07D2 D041 1846 CALR SKIPSP ! SKIP OVER BLANK & DON'T!
                                     ! UPDATE INPTR !
              1847
07D4 6B00 43B4 1848 DEC INPTR
07D8 0A08 0D0D 1849 CPB RLO, #ASCICR ! 1ST CHAR IN RLO, COMPARE!
07DC 9E08 1850 RET ! MITH CR !
             1851 END GET
07DE
             1852
```

```
1853
OZDE
              1854 GLOBAL GBFADR PROCEDURE
              1855
              1856 | **********************************
              1858 ! GENERATE ADDRESS OF RINGBF TO STORE OR GET NEXT CHAR
              1860 ! INPUT: RO = OFFSET TO RINGRE
              1861 1
              1862 ! OUTPUT: RO = OFFSET TO RINGRE FOR NEXT CHAR
              1863 ! R2 = ADDRESS OF RINGBF
              1844 1
              1867 ENTRY
              1868 LD R2, R0
07DE A102
07DE A102 1868 LD K2, K0
07E0 A900 1869 INC R0, #1 ! INC OFFSET !
07E2 0B00 0100 1870 CP R0, #RBSIZE ! MRAP AROUND IF HIT END!
07E6 EE01 1871 JR NZ, GEN1
07E8 8D08 1872 CLR R0 ! RESET OFFSET =0 !
07EA 0102 41B0
            1873 GEN1: ADD R2, #RINGBF ! CALCULATE ADDRESS!
07EE 9E08
              1874 RET
07F0
              1875 END GBEADR
              1876
              1877
07F0
              1878 GLOBAL MCZADR PROCEDURE
             1879
              1881 1
              1882 ! GENERATE ADDRESS OF MCZBUF TO STORE OR GET NEXT CHAR. !
             1883 !
            1884 ! INPUT: RO = OFFSET TO MCZBUF
             1885 1
            1886 ! OUTPUT: RO = UPDATED OFFSET
              1887 ! R2 = ADDRESS OF MCZBUF
              1888 1
             1890
             1891 ENTRY
07F0 A102
             1892
                      LD R2, RO
07F2 A900 1893
07F4 0B00 0100 1894
                      INC RO, #1 ! INC OFFSET !
                      CP RO, #RBSIZE ! WRAP AROUND IF HIT END!
07F8 EE01 1895 JR NZ, GENX
07FA 8D08 1896 CLR
                             RO ! RESET OFFSET =0 !
O7FC 0102 42BO 1897 GENX: ADD R2, #MCZBUF! CALCULATE ADDRESS!
             1898
0800 9E08
                       RET
0802
             1899 END MCZADR
             1900 ! PAGE EJECT !
             1901
0802
             1902 GLOBAL PTYINT PROCEDURE
             1903
```

```
1905 1
             1906 ! INTERRUPT INPUT ROUTINE, SAVE ALL USER REGISTERS.
             1907 | RECEIVE ONE CHARACTER FROM SERIAL LINE.
     1908 ! IF SYSTEM IS NOT IN TRANSPARENT MODE, THEN CHECK IF !
  1909 ! XOFF OR XON CHARACTER IS RECEIVED, SET OR RESET BIT
             1910 ! IN CFLAGS HORD.
             1911 1
             1912 ! OUTPUT : SET/RESET BIT IN CFLAGS IF RECEIVE XON/XOFF !
             1913 ! STORE CHARACTER IN RINGBF & UPDATE TTYPUT !
             1916
             1917 FNTRY
                             eR15, RO
             1918
0802 93F0
                       PUSH
0804 93F1
             1919
                       PUSH
                             @R15, R1
                             @R15, R2
0806 93F2
                     PUSH
             1920
             1921
             1922 ! READ CHARACTER AND CHECK FOR XON XOFF CHAR!
             1923
             1924 INB RL1, SIODB
0808 3A94 FF1R
             1925 RESB RL1, #PARITY! READ CHAR FROM TTY!
080C A297
             1926
080E 6700 43D4 1927 BIT CFLAGS, #TRNMDE ! DON'T CHECK XON, XOFF!
             1928
0812 FF18
                       JR
                             NZ, SUCHR ! IF IN TRANS, NDE !
0814 4A09 43BA 1929 CPB RL1, XONCHR ! RESET BIT IF XON!
             1930 JR NZ, CKOFF
0818 FF03
081A 6302 43B4
                     RES
                             CFLAGS, #OUTHLD ! RESET BIT SO ROUTINE !
            1931
             1932
                                      1 TYNR RESUME DUPUT 1
                      JR OUTS
081E E818
             1933
             1934 CKOFF:
             1935 CPB RL1, XOFCHR
1936 JR NZ, CKESC
0820 4A09 43BB
                                        ! SET BIT IF XOFF !
0824 FF03
0826 6502 43D4
             1937
                       SET CFLAGS, #OUTHLD ! SET BIT SO ROUTINE !
             1938
                             ! TYWR STOP OUTPUT!
082A E812
             1939 JR OUTS
             1940
             1941 CKESC:
082C 0A09 1B1B
           1942
                       CPB RL1,#ESCHAR
             1943 JR NZ, SVCHR
                                        ! NOT ESCAPE KEY !
0830 EE09
0832 6703 43D4 1944 BIT CFLAGS, #SNDMDE ! YES, CHECK IF IN SEND!
                       JR
0836 EE03
             1945
                             NZ, SETFG
0838 6704 43D4 1946 BIT CFLAGS, #LODMDE ! CHECK IF IN LOAD !
             1947 JR
                           Z, SVCHR
083C E603
                                         ! NO, JUST SAVE ESCAPE !
             1948 SETFG: SET CFLAGS, #ESCAPE
083E 6505 43D4
0842 E806
             1949 JR
                             OUTS
                                        ! SET BIT INDICATE ESCAPE!
                                         ! PRESSED IN LOAD/SEND !
             1950
             1951
             1952 SVCHR:
0844 6100 4304
             1953 LD
                             RO, TTYPUT
                      CALR GBFADR ! CALC ADDR OF RINGBF!
0848 D036
             1954
                     LD TTYPUT, RO ! UPDATE OFFSET !
084A 6F00 43C4 1955
```

```
LDB
                                  ! GET CHAR IN RINGBE!
084E 2E29
            1956
                           eR2, RL1
            1957
            1958 OUTS:
            1959 LD R1, $20038 ! BIT 0,1,2 MRITE REG 0 !
0850 2101 0038
            1960 ! BIT 3-5 RETURN FROM INTER!
0854 3A96 FF1D
                 OUTB SIOCA, RL1
            1961
            1962
            1963 POP R2, @R15
0858 97F2
085A 97F1
            1964 POP R1, @R15
                 POP
085C 97F0
            1965
                         RO, @R15
            1966 IRET
085F 7800
            1967
0840
            1968 END PTYINT
            1969 ! PAGE EJECT !
0860
            1971 GLOBAL MCZINT PROCEDURE
            1974 1
            1975 ! INTERRUPT ROUTINE FOR INPUT FROM MCZ (CHANNEL A)
            1976 ! RECEIVE ONE CHARACTER FROM SERIAL LINE.
   1978 ! OUTPUT : STORE CHARACTER IN NCZBUF & UPDATE NCZPUT !
     1981
           1982 ENTRY
0860 93F0
         1983 PUSH
                           @R15, R0
0862 93F1
            1984
                    PUSH @R15, R1
0864 93F2
            1985
                    PUSH @R15, R2
            1986
                 ! INPUT CHARACTER FROM SIO CHANNEL A !
            1987
            1988
0866 3A94 FF19 1989
                     IMB
                         RL1, SIODA
086A A297
                     RESB RL1, #PARITY ! READ CHAR !
            1990
            1991
086C 6100 43C8
           1992
                    LD
                          RO, MCZPUT
0870 D041
            1993
                    CALR
                          MCZADR ! CALC ADDR OF MCZBUF!
            1994
0872 6F00 43C8
                    LD MCZPUT, RO ! UPDATE OFFSET !
            1995 LDB @R2, RL1 ! SAVE CHAR IN MCZBUF!
0876 2E29
            1996
0878 2101 0038
            1997 LD R1, $%0038 ! BIT 0,1,2 MRITE REG 0 !
            1998
                                   ! BIT 3-5 RETURN FROM INTER!
087C 3A96 FF1D
            1999
                   OUTB SIOCA, RL1
            2000
            2001 POP
0880 97F2
                         R2, @R15
0882 97F1
                   POP
                         R1, @R15
            2002
0884 97F0
            2003
                    POP
                        RO, @R15
0886 7B00
            2004
                     IRET
            2005
0888
            2006 END MCZINT
            2007
```

```
2008
              2009 GLOBAL CHBSRC PROCEDURE
              2010
              2012 1
              2013 ! CHBSRC RESET SIO CHANNEL B FOR SPECIAL RECEIVE
              2014 ! CONDITIONS: PARITY ERROR, RX OVERRUN ERROR, FRAMING !
              2015 ! ERROR, END OF FRAME.
              2016 !
              2018
              2019 ENTRY
0888 93F1
              2020 PUSH @R15, R1
0888 93+1 2020 PUSH PRIS, R1
088A 2101 0030 2021 LD R1, #X30 ! BIT 3,4,5 ERROR RESET !
088E 3A96 FF1F 2022 OUTB SIOCB, RL1
0892 2101 0038 2023 LD R1, #X38 ! BIT 3,4,5 RETURN FROM !
0896 3A96 FF1D 2024 OUTB SIOCA, RL1 ! INTERRUPT !
0890 97F1 2025 POP R1, PRIS
089C 7B00 2026 IRET
089E
            2027 END CHBSRC
             2028
              2029
089E
             2030 GLOBAL CHASRC PROCEDURE
              2031
              2033 1
             2034 ! CHASRC RESET SIO CHANNEL A FOR SPECIAL RECEIVE
             2035 ! CONDITIONS: PARITY ERROR, RX OVERRUN ERROR, FRAMING!
             2036 ! ERROR, END OF FRAME.
             2037 !
      2038 | *********************************
    2039
2040 ENTRY
089E 93F1 2041 PUSH @R15, R1
08A0 2101 0030 2042 LD R1, #X30 ! BIT 3,4,5 ERROR RESET !
08A4 3A96 FF1D 2043 OUTB SIOCA, RL1
08A8 2101 0038 2044 LD R1, #X38 ! BIT 3,4,5 RETURN FROM ! 08AC 3A96 FF1D 2045 OUTB SIOCA, RL1 ! INTERRUPT! 08B0 97F1 2046 POP R1, @R15
08B2 7B00
        2047 IRET
08B4 2048 END CHASRC
             2049 ! PAGE EJECT !
2050
0884
             2051 GLOBAL PUTMSG PROCEDURE
        2052
```

515

```
2053 | **********************************
             2054 1
             2055 | PHT OUT MESSAGE POINTED BY R2, 1ST BYTE OF MESSAGES IS!
             2056 ! THE LENGTH OF THE MESSAGE IN BYTE
             2057 1
             2058 | INPUT: R2 = ADDRESS OF MESSAGES
             2059 1
             2060 ! OUTPUT: RO,R1,R2 ARE LOST
             2061 1
             2064 ENTRY
             2065 LD R1, #CONOBF
08B4 2101 4130
                      CLR RO
08B8 8D08
             2066
             2067 LDB RLO, GR2 ! # OF BYTE OF MESSAGE!
08BA 2028
                      ADD R1, RO
08BC 8101
             2068
             2069 LD OUTPTR, R1
088F AF01 438A
             2070
             2071 INC R2
2072 LD R1, #CONOBF
2073 LDIRB @R1, @R2, R0 ! TRANSFER DATA TO CONOBF!
08C2 A920
0804 2101 4130
08C8 BA21 0010
08CC 5E08 0982
             2074 JP PUTNCR
             2075
             2076 END PUTMSG
0800
             2077
             2078
             2079 GLOBAL LOD2 LABEL
             2080 GLOBAL LOD3 PROCEDURE
             2083 1
             2084 ! LOD3 EXPECTS THREE ARGUMENTS AFTER THE COMMAND IN
             2085 ! CONIBF BUFFER. THE ASCII ARGUMENTS ARE CONVERTED
             2086! TO HEX ADDRESSES.
             2087 ! LOD2 EXPECTS TWO ARGUMENTS AFTER THE COMMAND.
             2088 !
             2089 ! OUTPUT : (LOD3) R5 - 1ST ARGUMENT, R4 - 2ND ARGUMENT
             2090 ! R3 - 3RD ARGUMENT
             2091 !
                         (LOD2) R4 - 1ST ARGUMENT, R3 - 2ND ARGUMENT !
             2092 !
                         RETURN C FOR INSUFFICIENT ARGUMENTS
             2094 1
                         RETURN NC FOR RECEIVING CORRECT ARGUMENTS
             2096 ! *** RO, R2 ARE LOST ***
             2099
             2100 ENTRY
08D0 D0C5
             2101
                       CALR
                             NXTARG ! SKIP COMMAND TO NEXT CHAR!
             2102
                      CALR GETADN
08D2 D0D9
                                       ! GET 1ST ARGUMENT IN R3 !
08B4 E608
             2103
                      JR Z, ERNS ! CAN'T BE CR, EXPECT MORE!
08D6 A135
             2104
                      LB
                             R5, R3
```

```
2105 JR LX2
08B8 F801
          2106
          2107 LOD2:
OSDA DOCA
          2108
                  CALR NXTARG ! SKIP OVER COMMAND!
          2109 LX2:
          2110 CALR GETADN ! GET 2ND ARGUMENT !
ORDE DODE
          2111 JR Z , ERNS ! ERROR, CR AFTER 2ND ARGU.!
2112 LD R4,R3 ! SAVE 2ND ARGUMENT !
08DE E603
08E0 A134
          2113
          2114 CALR GETADN ! GET 3RD ARGUMENT !
08F2 D0F1
          2115
                  RET NC ! GOT ARGUNENT !
08E4 9E0F
          2116
08FA 8D81
          2117 ERNS: SETFLG C
                             ! RETURN C FOR ERROR !
08E8 9E08
          2118
                  RET
                              ! ARGUMENT MISSING !
          2119
          2120 END LOD3
OSFA
      2121
          2122
        2123
08FA
          2124 GLOBAL PUTRG2 PROCEDURE
          2125
          2128 ! PUTRG2 PRINTS THE CONTENTS OF REGISTER 14, 15
          2129 ! CURRNET PC AND FCW
          2130 !
          2132 ENTRY
   2133
                LDK R6, #4 ! # OF REGISTER !
08EA BD64
          2134
08EC 2101 43F2
          2135 LD R1, #R14_ ! ADDRESS OF R14 !
          2136 JR DL2P
08F0 E814
          2137
08F2
          2138 END PUTRG2
          2139
          2140
          2141
          2142 GLOBAL RHED1 PROCEDURE
08F2
        2146 ! RHED1 PRINT REGISTER 0-13 HEADER
          2147 !
          2148 ! *** RO, R1, R2 ARE LOST
          2151
          2152 ENTRY
          2153 LDK RO,#14 ! PRINT LITERAL RO-R14 !
OBF2 BDOE
                 LD R5, #LRO ! REG LITERAL TABLE ADDR !
08F4 2105 0E7C
          2154
          2155
08F8 6102 43B6 2156 DLOP: LD R2, OUTPTR ! CURRENT CONOBF ADDR!
```

```
08FC 0C25 5252 2157 LDB @R2, #'R'
          2158
0900 6900 43B6 2159 INC OUTPTR
0904 2151 2160 LD R1, GR5 ! GET LITERAL!
2164
2165 INC
090C 6901 43B6 2165
                       OUTPTR, #2 ! ADD 2 SPACE !
0910 A951 2166 INC R5, #2 ! ADVANCE LIT TABLE ADDR!
      2167 JR DLOP
0912 EBF2
          2168 END RHED1
          2169
          2170
0914
          2171 GLOBAL PUTRG1 PROCEDURE
          2172
          2175 ! PUTRG1 PRINTS CONTENTS OF REGISTER 0 TO 13
          2176 1
          2178
          2179 ENTRY
0914 BD6E 2180 LDK R6, #14 ! # OF REGISTER!
0916 2101 43D6 21B1 LD R1, #RO_ ! REGISTER CONTENT ADDRESS ! 091A 2115 2182 DL2P: LD R5, @R1 ! CONVERT REGISTER CONTENT !
091C DFF6 2183 CALR PUT2! TO FOUR 8-BIT ASCII CHAR!
          2184 DEC R6, #1 ! CHECK IF DONE!
2185 JR Z, PUTPTR ! PRINT CONOBF, RETURN!
091E AB60
0920 E62A
       2186
0922 6900 43B6 2187 INC OUTPTR, #1 ! SPACE !
          2188 INC R1, #2 ! NEXT REG CONTENT ADDRESS!
0926 A911
              JR DL2P
0928 EBF8
          2189
          2190
092A
          2191 END PUTRG1
          2192
          2193
092A
          2194 GLOBAL RHED2 PROCEDURE
          2195
          2197 1
          2198 ! RHED2 PRINTS THE HEADER FOR REG 14, REG 15, PC & ECN !
          2201
          2202 ENTRY
          2203 LDK RO, #4 ! # OF REGISTER !
092A BB04
092C 2105 0E98
         2204
                LD R5, #LR14 ! ADDRESS OF LITERAL!
0930 E8E3
          2205 JR DLOP
0932
          2206 END RHED2
          2207
          2208
```

```
2209 GLOBAL PUT1 LABEL
               2210
0932
               2211 GLOBAL PUT2 PROCEDURE
               2212
               2214 1
               2215 | PUT2 CONVERTS FOUR 4 BITS HEX VALUE IN R5 TO FOUR
               2216 ! 8 BIT ASCII CHARACTER AND STORE THEM IN CONOBF.
               2217 ! OUTPTR ISINCREMENTED BY 4 WHICH POINTED TO THE NEXT !
               2218 ! EMPTY BUFFER
               2219 ! EACH 4 BIT HEX VALUE IS ALSO ADDED TO AN ACCUMULATOR !
               2220 ! RL3 BEFORE THEY ARE CONVERTED TO ASCII CHARACTERS
               2221 ! (USED BY LOAD/SEND, PUNCH/TAPE)
               2222 1
               2223 | PUT1 CONVERTS THO 4-BITS HEX VALUE IN RL5
               2224 1
               2225 ! INPUT: R5 = 16 BIT VALUE (PUT2)
               2226 ! RL5= 8 BIT VALUE (PUT1)
               2227 1
                           RL3= B BIT ACCUMULATOR
               2228 1
               2229 ! OUTPUT: RL3= UPDATED ACCUMULATOR
               2230 ! TWO OR FOUR ASCII CHARACTERS STORED IN CONOBF!
                          OUTPTRINGREMENT BY 2 OR 4
               2231 1
               2232 1
               2233 ! *** RO IS LOST ***
               2236
               2237 ENTRY
0932 A050
               2238
                         LDB RHO, RHS ! CONVERT 1ST BYTE !
0934 DFFF
               2239
                         CALR
                               PUTA
               2240
0936 A0D0
               2241 PUT1:
                         LDB
                                RHO, RL5
                                           ! CONVERT LONER BYTE
              2242
0938 BE08
               2243 PUTA:
                         RIDB
                                RLO, RHO
                                         ! CONVERT 4 BITS/ TIME !
093A DFFF
               2244
                         CALR
                                INVCON
093C BE08
               2245
                         RLDB
                                RLO, RHO
                                           ! NEXT 4 BITS
               2246
093E 0608
               2247 INVCON: ANDB
                                RLO, #%OF
                                           ! ONLY USE 4 BIT
0942 808B
               2248 ADDB
                               RL3, RLO ! ADD VALUE TO ACCUMULATOR!
              2249
0944 0A08
              2250
                         CPR
        OAOA
                                RLO, #%0A
                                          ! CK FOR 10-15 VALUE
0948 E702
                         JR
              2251
                                C, NOADD
094A 0008 0707
              2252
                         ADDB
                                RLO, #7
                                          ! CHANGE TO ALPHA
094E 0008 3030
              2253 NOADD: ADDB
                                RLO, $%30 ! CONVERT TO ASCII
              2254
0952 93F1
              2255 PUSH
                               @R15, R1
0954 6101 43B6
              2256
                         LD
                                R1, OUTPTR ! CONOBE OFFEST
0958 2E18
              2257
                         LDB
                                eR1, RLO
095A 6900 43B6
                                OUTPTR
              2258
                         INC
095E 97F1
              2259 POP
                               R1, @R15
0960 9E08
              2260
                         RET
```

```
2241
              2262 END PUT2
0962
              2263
              2264
              2265
0962
              2266 GLOBAL NULL PROCEDURE
              2267
              2268 | *********************
              2270 ! NULL OUTPUT A SEQUENCE OF NULL CHARACTERS. NULLCT IS
              2271 ! THE COUNT OF NULL CHARACTER.
              2272 !
              2274
              2275 ENTRY
              2276 TEST
0962 4B04 43BC
                               NULLCT
0966 9E06
              2277
                        RET
                               7
                                           ! NO NULL CHAR !
                       LD
0968 6101 43BC
              2278
                               RI, NULLCT ! PUT OUT NULL CHAR !
                   CLR
096C 8D08
              2279
                               RO
096E DOFD
              2280 SNULL: CALR
                             TYNR
              2281 ! DJNZ
                               RI, SNULL !
0970 AB10
              2282
                       DEC
                              R1
              2283 JR
0972 EEFD
                               NZ, SNULL
              2284
                      RET
0974 9E08
              2285
0976
              2286 END NULL
              2287
              2288
              2289
              2290 GLOBAL PUTNER LABEL
0976
              2291 GLOBAL PUTPTR PROCEDURE
              2293 ! ****************************
              2294 !
              2295 ! PUTPTR OUTPUT CHARACTRS IN CONOBE TO TTY WITH CR
              2296 ! ADDED TO THE END OF CHARACTER STREAM. OUTPTR IS THE !
              2297 ! ADDRESS OF CONOBF CONTAINING LAST CHARACTER.
              2299 ! PUTNCR OUTPUT CHARACTER STREAM IN CONOBF.
              2300 1
              2301 ! INPUT: CONOBF - BUFFER WITH CHARACTERS
              2302 !
                          OUTPTR - CONOBF POINTER OF THE LAST CHAR.
              2303 1
              2304 ! OUTPUT: OUTPTR=#CONOBF, CONOBF BUF FILL WITH BLANK
              2305 !
              2306 ! *** RO, R1, R2 ARE LOST ***
              2308 | ********************************
              2309
              2310 ENTRY
              2311 LD R2, OUTPTR
0976 6102 43B6
097A 0C25 0D0D 2312
                        LDB @R2, #ASCICR ! STORE CR IN CONOBF !
```

```
097E 6900 43B6 2313 INC OUTPTR, $1 ! ADVANCE CONOBE PTR !
             2314
0982 2101 4130 2315 PUTNCR: LD R1,#CONOBF
             2316
0986 2018
             2317 OUTLOP: LDB RIO, MR1
0988 A910
             2318 INC R1
098A D10B 2319 CALR TYWR ! PUT OUT CHAR!
098C E604 2320 JR Z, PUTLF ! NO, CK IF PUT OUT CR!
098E 4B01 43B6 2321 CP R1, OUTPTR ! CHECK IF ALL DONE!
             2322 JR C, OUTLOP ! NOT YET !
2323 JR BINT | DONE, RESET POINTED |
0992 E7F9
                      JR BINT
0994 F804
           2324
             2325 PUTLF:
             2326 LDB RLO, #ASCILF ! YES, ADD LF!
0996 C80A
0998 D112
             2327
                      CALR
                            TYNR
2328
099A 5F00 0962' 2329
                                  ! PRINT NULL CHAR !
                      CALL
                            NULL
099E 4D05 43B6
             2330 BINT: LD OUTPTR, #CONOBF! RESET CONOBF POINTER!
09A2 4130
09A4 2102 0040 2331 LD R2, #COSIZE/2 ! INIT CONOBE WITH BLANK !
09A8 4D05 4130 2332 LD CONOBF, #' '
09AC 2020
09AE 2100 4130
            2333 LD RO, #CONOBF
09B2 2101 4132 2334 LD R1, #CONOBF+2
09B6 BB01 0210 2335 LDIR @R1,@R0, R2
             2336
                     RET
09BA 9E08
             2337
             2338 END PUTPTR
09BC
             2339
             2340 ! PAGE EJECT !
             2341
09BC
             2342 GLOBAL IOPORT PROCEDURE
             2345 !
             2346 ! PORTRN PERFORMS PORT READ & MRITE FUNCTIONS EITHER !
             2347 ! IN BYTE OR WORD, DEFAULT TO BYTE READ & WRITE
             2348 !
             2349 ! SYNTAX: PORT (PORT ADDR.) [N:B]
        2350 1
       2352
   2353 ENTRY
OPBC D13B 2354 CALR NXTARG
09BE 5E06 014E 2355 JP Z, EROR ! NO ADDRESS!
09C2 D151 2356 CALR GETADN
09C4 A136
             2357 LD R6, R3 ! SAVE PORT ADDR! 2358 JR Z, GOTCR
                    JR
09C8 0A08 5757 2359 CPB RLO, *'W' ! NEXT CHAR IN RLO!
09CC E61B 2360 JR Z, PRNORD ! READ IN WORD!
09CE 0A08 4242 2361 CPB RLO, *'B'
09D2 5E0E 014E 2362 JP NZ, EROR
```

```
2363 ! INDICATOR FOR BYTE/WORD!
    2363
0906 8078
0908 3069
               2365 INB RL1, QR6 ! READ DATA IN BYTE!
               2366
               2367
                    I DISPLAY PORT ADDRESS & THE DATA I
               2368 ! R7=0 READ BYTE, R7=1 READ WORD!
               2369
09DA A165
               2370 PX: LD R5, R6
              2371 CALR PUT2 ! OUTPUT PORT ADDRESS!
2372 INC OUTPTR
2373 LD R5, R1
2374 TEST R7 ! OUPUT DATA BYTE OR HC
2375 JR Z, PUTBYT
09DC D056
09DE 6900 43B6
09E2 A115
                                           ! OUPUT DATA BYTE OR WORD!
09F4 8N74
09E6 E602
         2376 CALR PUT2
2377 JR POUT
09E8 D05C
09EA E801
               2378 PUTBYT: CALR PUT1
09EC D05C
     2379 POUT:
09EE 6900 43B6 2380 INC OUTPTR
09F4 DIDO 2382 GET C 09F6 9E07 2383 RET C 09F8 8D74 2384 TEST R7 09FA E602 2385 JR Z, MBYTE 09FC 3F63 2386 OUT QR6, R3 09FE 9E08 2387 RET 0A00 3E6B 2388 MBYTE: OUTB QR6, RL3 0A02 9E08 2389 RET
               2382 CALR GETNUM ! GET NEW DATA !
2383 RET C ! RECEIVE 'Q' !
2384 TEST R7
2385 JR Z, WBYTE
2386 OUT @R6, R3 ! REPLACE WITH NEW CONTENTS!
2387 RET
               2390
               2391 PRWORD: LDK R7, #1 ! READ DATA IN WORD!
0A04 BD71
               2392 IN R1, @R6
0A06 3D61
               2393 JR
2394
0A08 E8E8
                                PX
               2394
               2395 END TOPORT
               2396 ! PAGE EJECT !
               2397
OAOA 2398 GLOBAL PUNCH PROCEDURE
               2402 ! PUNCH IS USED TO PUNCH A PAPER TAPE COPY OF THE
               2403 ! PROGRAM IN MEMORY FROM THE STARTING ADDRESS TO
               2404 ! ENDING ADDRESS SPECIFIED BY (BEG.ADDR) & (END ADDR) !
               2405 ! THE PUNCH WILL BE TURNED ON AUTOMATICALRY, AND THE !
               2406 ! PROGRAM WILL BE PUNCHED IN TEKTRONIX FORMAT. !
               2409 ENTRY
2412 LD R8, R4 ! STARTING ADDR!
0A10 A148
            2413 LD R7, R3 ! ENDING ADDR ! 2414 SUB R7, R8
0A12 A137
0A14 8387
```

```
0A16 5E07 014E 2415 JP C, EROR ! ENDING ADDR TOO SMALL ! 0A1A A970 2416 INC R7
             2417
0A1C C812
             2418
                       IDR
                              RLO, #TAPON ! PUCH TAPE ON CHAR !
DATE DISS
            2419
                       CALR
                             TYNR
0A20 DFBC 2420
                       CALR
                             NULLTP ! TAPE LEADING CHAR !
             2421
0A22 0B07 001E 2422 NXTRC: CP
                             R7, #30
                             NC, S30 ! SEND 30 BYTE ! R9, R7 ! SEND ( 30 BYTE !
0A26 EF03 2423
                       .IR
0A28 A179
             2424
                       LD
           2425
                             R7
0A2A 8D78
                       CLR
                             RSDD
                       JR
0A2C E804
            2426
                       LD R9, #30
            2427 $30:
0A2F 2109 001F
0A32 0307 001E 2428
                       SUB R7, #30
              2429
0A36 2102 0976' 2430 RSDD: LD
                             R2, #PUTPTR ! SEND DATA TO TERMINAL!
OA3A DEE6 2431 - CALR RECSND
OA3C EEF2 2432 JR NZ, NXTRC ! SEND MORE !
             2433
         2434 CLR R6
0A3F 8D68
                                    ! ADDRESS = 0 !
                       LD R2, #PUTPTR
CALR LASREC ! SEND LAST RECORD-TERMINAL!
CALR NULLTP ! LEADING CHAR AT TAPE END!
0A40 2102 0976' 2435 LD
0A44 DED2 2436
0A46 DFCF 2437
           2438 LDB RLO, #TAPOFF ! PUNCH TAPE OFF !
0A48 C814
             2439
2440
                       CALR TYNR
0A4A D16B
0A4C 9E08
                       RET
             2441
2442 END PUNCH
0A4F
              2443 ! PAGE EJECT !
           2444
              2445 GLOBAL TAPE PROCEDURE
0A4F
              2446
              2447 | *********************************
              2448 !
              2449 | TAPE LOADS A PAPEER TAPE AND STORES DATA INTO MEMORY !
              2450 ! TURN ON THE READER AND THE DATA ON TAPE IS IN
              2451 ! TEKTRONIX FORMAT. AFTER THE END OF TAPE, TURN OFF !
     2452 ! THE TAPE & MESSAGE PRINTED
              2453 !
              2455 ENTRY
              2456
           2457 LDB RLO, #XONC
0A4F C811
           2458 CALR TYWR ! SEND XON TO START TAPE !
0A50 D16E
              2459
              2460 ! WAIT FOR A '/' TO BEGIN STORING CHARACTER INTO !
        2461 ! CONIBF. CONIBF DOES NOT INCLUDE '/'. !
        2462
            2463 WATCT:
OA52 D17B 2464 CALR TYIN
            2465 CPB RLO, #'/' ! WAIT UNTIL '/'!
0A54 0A08 2F2F
OA58 EEFC
            2466 JR NZ, WATCT
```

```
2467
               2468 LD
0A5A 2104 40B0
                                 R4, #CONIBE
OASE CRSO
               2469
                        IDR
                                 RL3, #80
                                            I MAX GET BO CHARACTER!
               2470
               2471
               2472 ! START STORING CHARACTER, IGNORE CONTROL CHAR!
               2473
                    ! ALSO CHECK FOR CARRIAGE RETURN FOR END OF RECORD !
               2474
0A60 D182
               2475 LCO:
                          CALR
                                 TYTH
               2476
0A62 2E48
               2477
                                 PR4, RLO | STORE CHAR |
                          LDB
                                 RLO, #ASCICR ! CHECK FOR END OF RECORD !
0A64 0A08 0D0D
               2478
                          CPR
                                 Z, SXOFC ! YES, STOP THE TAPE !
RLO, * ' ! IGNORE CONTROL CHAR!
0A68 F608
               2479
                          .IR
0A6A 0A08 2020 2480
                         CPB
OAGE ESF8
               2481
                         JR
                                 MI,LCO
               2482
                                 R4, $1 | GOOD CHARACTER |
0A70 A940
                         INC
0A72 FB0A
               2483
                          DBJNZ RL3, LCO ! DECREMENT CHAR COUNT!
               2484
               2485
               2486 ! ALREADY RECEIVE 80 CHARACTERS, SKIP OVER ALL !
               2487
                     ! CHARACTERS UNTIL CARRIAGE RETURN. !
               2488
0A74 D18C
               2489 LOC2:
                         CALR
                                 TYIN
OA76 OAO8 ODOD
                          CPB
               2490
                                 RLO, #ASCICR ! REST OF CHAR EXCEPT CR !
OA7A EEFC
               2491
                          JR
                                 NZ, LOC2
               2492
0A7C 6E08 4100
                          LDB
                                 CONIBF+80, RLO ! GOT CR !
               2493
               2494
               2495 SXOFC:
0A80 C813
               2496
                          LDB
                                 RLO, #XOFFC ! SEND XOFF TO STOP TAPE !
0A82 D187
               2497
                          CALR
                                 TYMR
               2498
               2499
0A84 DF32
                          CALR
                                 REKCKS
                                 C, BDATA
0A86 E704
               2500
                                            ! CHECKSUM BAD !
               2501
               2502
                                 RH3 ! CHECK IF LAST RECORD!
0A88 8C34
                          TESTR
                                         I YES !
0A8A E606
               2503
                          JR
                                 Z, TAPFIN
                                 STREC ! STORE DATA IN MEMORY !
OASC DEEF
               2504
                          CALR
OASE ESDF
                          JR
               2505
               2506
               2507 BDATA:
                          CALR
0A90 D179
               2508
                                 CRLF
                          LD
                                 R2, #BADAT ! MESSAGE FOR BAD DATA!
0A92 2102 0E10
               2509
               2510
                                PUTMSG ! ABORT & BACK TO DEBUG !
0A96 D0F2
                          CALR
               2511
               2512 TAPFIN:
                         LD R1,#XFFFF ! WAIT FOR A WHILE BEFORE !
0A98 2101 FFFF
               2513
OA9C AB10
                                R1,#1
                                            ! EMPTY INPUT BUF, IN !
               2514 WATT:
                          DEC
                                            ! CASE RECEVIED EXTRA CHAR!
OA9E EEFE
               2515
                          JR
                                 NZ, WATT
               2516
                                             ! BEFORE XOFF SEND OUT !
              2517 CLR TTYPUT
0AA0 4D08 43C4
                          CLR TTYGET ! CLEAR INPUT BUFFER !
0AA4 4D08 43C6
             2518
```

```
2519 RET 2520
OAAR 9FOR
              2521 END TAPE
              2522
              2523
              2524 GLOBAL NULLTP PROCEDURE
              2527 1
              2528 | NULL TP PUNCH 50 WILL CHARACTER FOR TAPE LEADER.
              2531 ENTRY
             2532 LDB RL1, #50
2533 CLR RO
2534 LRS: CALR TYNR
0AAA C932
OAAC 8D08
DAAE D19D
              2535 DBJNZ RL1, LRS
2536 RET
0AR0 F902
0AR2 9F08
              2537 END NULLTP
              2538 ! PAGE EJECT !
              2539
              2540 GLOBAL LODCHD PROCEDURE
OAB4
              2542 | *******************************
              2543 1
              2544 ! LOAD COMMAND EXPECT RECORDS OF DATA (ASCII CHARACTER) !
              2545 ! IN THE FOLLOWING FORMAT:
              2546 ! (ADDRESS(4)) (COUNT(2)) (CKSUM1(2))(DATA(2)) (DATA(2))!
              2547 ! (DATA(2)) ... (DATA(2)) (CKSUM2(2))
              2548 !
              2549 ! THE THO CHECKSUM VALUES ARE VERIFIED BEFORE DATA ARE !
              2550 ! STORED IN MEMORY SPECIFIED BY THE (ADDRESS).
              2551 ! (CKSUM1) IS THE CHECKSUM FOR THE (ADDRESS) & (COUNT).!
              2552 ! (CKSUM2) IS THE CHECKSUM FOR THE DATA PORTION OF THE !
              2553 !
                          RECORD.
              2554 ! IF THE RECORD RECEIVED WITH COUNT=0, INDICATES THE !
              2555 ! END OF THE LOAD DATA.
              2556 ! AFTER RECEIVING EACH RECORD, EITHER AN ACKNOWLEDGE !
              2557 ! (ASCII O FOLLOWED BY CR) OR NON-ACKNOWLEDGE (ASCII 7 !
              2558 ! FOLLOWED BY CR WILL BE SENT.
              2559 !
              2560 ! IF RECORD RECEIVED IS BEGIN WITH //, SEND THE CHAR !
              2561 ! STRING AFTER // TO TERMINAL, AND ABORT THE COMMAND. !
              2562 !
              2563 ! SYNTAX: LOAD (FILENAME)
              2564 !
              2566
              2567 ENTRY
              2568
              2569
                     ! RESET MCZ INPUT BUF POINTER, INSURE FILENAME EXIST!
              2570
```

```
2571 CALR CKFNAM
OAB4 DF71
               2572
               2573
                      ! SEND COMMAND TO MCZ/ZDS TO START THE ZBO PROGRAM!
               2574
                     ! AND CHECK IF PROGRAM GET LOADED.
               2575
0AB6 6504 43D4
               2576
                         SET
                                CFLAGS, #LODMDE ! SET BIT FOR LOAD COMMAND!
OARA DED1
               2577
                         CALR
                                SNDCMD
                              Z ! PROGRAM DIDNOT GET LOADED!
OABC 9E06
               2578
                         RET
               2579
               2580 NXXTR:
OABE DE69
               2581
                         CALR
                                LODREC
                                           ! FETCH RECORD INTO CONIBE!
                       LD
OACO 2102 4080
               2582
                                R2, #CONTRE
0AC4 2028
               2583
                      LDB
                                RLO, @R2
                                           ! CHECK IS ERROR RECORD !
OAC6 0A08 2F2F
               2584
                        CPR
                                RLO, $1/1
OACA EEOB
               2585
                        JR
                                NZ, RECOK ! NO !
               2586
                         LD R1, #CONOBF
OACC 2101 4130
               2587
OADO 2103 0020
               2588
                                R3, $%20 ! MAX 40 CHAR MESSAGE !
                         111
              2589
2590
                                eR1, eR2, R3 ! PRINT ERROR MESSAGE !
OAD4 BB21 0310
                        LDIR
0AD8 4D05 43B6
                        LD
                                OUTPTR, #CONOBF+%20
OADC 4150
OADE 5E08 0982
                         JP
                                PUTNCR ! & ABORT LOAD !
               2591
               2592
               2593 RECOK:
OAE2 6705 43D4
               2594
                         BIT
                                CFLAGS, #ESCAPE ! CHECK IF ESCAPE KEY HIT !
                      JR
QAE6 EE22
               2595
                                NZ, LABORT ! YES, ABORT !
               2596
               2597 CALR
OAES DE64
                                REKCKS ! VERIFY TWO CHECKSUMS & !
               2598 ! RETURN COUNT IN RH3 !
OAEA EF02
                                NC,LODOK ! GOOD CHECKSUM !
               2599
               2600
               2601
                       CALR
                                BADCKS ! BAD CKSUM, SEND NON-ACK !
OAEC DFOF
               2602
                       JR
                                NXXTR ! TRY AGAIN
OAEE E8E7
               2603
               2604
               2605
                    ! CHECK IF THIS IS THE LAST RECORD (COUNT=0) !
              2606
                      ! PRINT ENTRY POINT OF PROGRAM JUST LOADED FROM MCZ !
              2607
              2608 LODOK:
                    TESTB RH3 ! CHECK COUNT IN RECORD !
0AF0 8C34
              2609
                         JR
                                NZ, STRED ! NOT O, STORE DATA IN MEMORY !
OAF2 EE12
              2610
                                GODCKS ! SEND ACKNONLEDGE !
                    CALR
OAF4 DFOF
              2611
              2612
OAF6 5400 40B0
              2613
                        LDL
                                RRO, CONIBF ! MOVE 4 ASCII CHAR FOR !
OAFA 5D00 413C
                        LDL
             2614
                                CONOBF+%OC, RRO ! ENTRY POINT !
OAFE 2100 0E70
             2615
                        LD
                                RO, BENTR
              2616 LD
OBO2 2101 4130
                                R1, #CONOBF ! MOVE 'ENTRY POINT' TO !
0B06 2102 0006
              2617
                        LD
                                R2,#6
                      LDIR
OBOA BB01 0210
              2618
                                @R1,@R0,R2
OBOE 4D05 43B6 2619
                       LD
                                OUTPTR, #CONOBF+%10
OB12 4140
OB14 DODO
              2620
                      CALR PUTPTR
```

| OB16 | 9E08 | | 2621 | | RET | | ! | PRINT 'ENTRY POINT XXXX' ! | |
|-------------|------|------|------|----------|---------|-------------|---------|-----------------------------|--|
| | | | 2622 | | | | | | |
| | | | 2623 | STRED: | | | | | |
| 0818 | DF2C | | 2624 | | CALR | RECADR | 1 | VERIFY STARTING ADDR IN ! | |
| OB1A | OBO1 | 4460 | 2625 | | CP | R1,#USERAM | 1 | RECORD GREATER THAN 4460! | |
| OBIE | ED03 | | 2626 | | JR | PL, SAVDAT | 1119 | Blon-1-12AC | |
| 0B20 | 2102 | 0E1A | 2627 | | LD | R2,#ADBMSG | 1 | ABORT, PRINT 'NRONG ADDR' ! | |
| 0B24 | E805 | | 2628 | | JR | SMS6 | | MON T FERC | |
| | | | 2629 | | | 94 (99) | | | |
| | | | | SAVDAT: | | | | | |
| 0B26 | DF28 | | 2631 | | CALR | GODCKS | 1 | SEND ACKNOWLEDGE ! | |
| 0828 | DF3D | | 2632 | | CALR | STREC | 1 | STORE DATA IN MEMORY ! | |
| OB2A | E8C9 | | 2633 | | JR | NXXTR | 11 | DONE, GET NEXT RECORD ! | |
| | | | 2634 | | | n Zelepyn | | Tak bake | |
| OB2C | 2102 | 0E68 | | LABORT: | LD | R2, #ESCMSG | | | |
| 0830 | D13F | | 2636 | SMSG: | CALR | PUTMS6 | !! | ISER WANT TO ABORT ! | |
| 0B32 | DF30 | | 2637 | | CALR | ABKCKS | 1 5 | GEND '9' TO MCZ ! | |
| | | | 2638 | | | | | AND RETURN TO DEBUG! | |
| 0B34 | 9E08 | | 2639 | | RET | | THE RES | 715 1 42 kg | |
| | | | 2640 | | | | | | |
| 0B36 | | | | END LODG | CMD | | | | |
| | | | 2642 | ! PAGE ! | EJECT ! | | | | |
| | | | 2643 | | | | | | |
| 0B36 | | | | | | | | | |
| | | | 2645 | | | | | | |
| | | | | | | | | | |

```
2647 1
                2648 ! SEND TRANSFER DATA IN MEMORY TO MCZ/ZDS SYSTEM.
                2649 ! ALL DATA ARE CONVERTED TO ASCII CHARACTER BEFORE
                2650 | THEY ARE SEND. IT SEND MAXIMUM 30 BYTES OF DATA IN
                2651 ! ONE RECORD IN THE FOLLOWING FORMAT:
                2652 1
                2653 ! /(ADDRESS) (COUNT) (CKSUM1) (DATA) (DATA)...(DATA) !
                2654 1 (CKSUM2) CR
                2655 ! WHERE (ADDRESS) - THE ADDRESS OF FIRST BYTE OF DATA !
                2656 ! (COUNT) - # BYTE OF DATA IN THAT RECORD
                2657 ! (CKSUMI) - CKSUM OF (ADDRESS) & (COUNT)
                2658 ! (DATA) - DATA BYTE OF SPECIFIED MEMORY
                            (CKSUM2) - CHECKSUM OF THE DATA BYTES
                2659 1
                2660 1
                2661 ! AFTER SENDING ALL DATA, A RECORD IN THE FOLLOWING
                2662 ! FORMAT WILL BE SEND AT THE END:
                2663 1
                2664 ! /(ENTRY ADDRESS) OO (CKSUM) CR
                2665 | WHERE CKSUM - CKSUM OF ENTRY ADDRESS
                2666 1
                2667 ! SEND COMMAND WILL RETRY 10 TIMES IF NON-ACKNOWLEGE IS !
                2668 ! RECEIVED. THEN IT WILL SEND //ERROR MESSAGE CR
                2669 ! TO ABORT COMMAND
                2670 !
                2671 ! SYNTAX: SEND (FILENAME) (BEGINNING ADDRESS) (ENDING
                2672 1
                             ADDRESS) [(ENTRY ADDRESS)]
                2674 ! **********************************
                2675
                2676 ENTRY
                2677
                2678
                       ! RESET MCZ INPUT BUFFER POINTER, INSURE FILENAME EXIST!
                2679
OB36 DFB2
                2680
                            CALR
                                   CKFNAM
                2681
                2682
                           CALR
                                   LOD2
                                               ! GET NEXT 2 ARGUMENT IN !
OB38 D130
                2683
                                               1 R4, R3
0B3A E705
                2684
                           JR
                                   C, SENDER
                                              ! ERROR, SEND MESSAGE !
                2685
OB3C A148
                2686
                           In
                                   R8, R4
                                               ! SAVE STARTING ADDRESS
                                   R7, R3
083E A137
                2687
                           LB
                                               ! SAVE ENDING ADDR
                2688
0840 D210
                            CALR
                                   GETADN
                                               ! CHECK IF ANY ENTRY ADDR !
                2689
                           LD
0B42 A136
                2690
                                   R6, R3
                                               ! IF NO ENTRY ADDR, DEFAULT!
                                               ! TO ZERO
                2691
                2692
OB44 8387
                            SUB
                                   R7, R8
                                               I FIGURE # OF BYTES
                2693
0B46 5E07 014E
                2694 SENDER: JP
                                   C, EROR
                                               ! ERROR IN ADDRESSES !
                                               ! SEND BEG.ADDR TO END ADDR!
OB4A A970
                2695
                            INC
                                               ! INCLUSIVE
                2696
                            SET CFLAGS, #SNDMDE ! SET BIT IN SEND PROCESS!
OB4C 6503 43D4 2697
```

```
2698
   2699 I SEND COMMAND TO MCZ/ZDS TO START THE ZBO PROGRAM I
 2700 ! AND CHECK IF PROGRAM GET LOADED.
             2701
OBSO DE1C
             2702
                     CALR SNDCHD
0B52 9E06
          2703 RET Z ! PROGRAM DIDNOT GET LOADED!
             2704
              2705
                  ! NAIT FOR ACKNONLEDGE, ALSO CHECK IF ESCAPE HIT!
            2706
             2707 CALR
2708 JR
0854 DE36
                             WATACK
                                      ! WAIT FOR ACKNOWLEDGE FIRST!
                             Z, OPOK
0B56 F603
             2708
                      JR
                                      ! FILE OPEN OK !
OB58 2102 OE56 2709 LD R2,#OPNMSG ! GOT '7' OR '9' FROM MCZ FOR!
                             PUTM ! UNABLE TO OPEN FILE !
OR5C F831
             2710
                  JR
            2711
             2712 OPOK:
085E 6705 43D4 2713 BIT
                             CFLAGS, #ESCAPE
        2714 JR Z,NXTR
0B62 E605
             2715 SABTS:
             2716 CALR
OB64 DEB4
                             SNDERR ! SEND ERROR RECORD TO MCZ!
0866 2102 0E68 2717
                      1 11
                             R2, #ESCMSG ! 'ABORT' TO TERMINAL !
OB6A D15C 2718 CALR PUTMSG
086C 9F08
             2719
                      RET
         2720
         2721 ! DETERMINE THE VALUE FOR (COUNT) IN THE NEXT RECORD!
  2722 ! R8 = STARTING ADDRESS, R7 = # OF BYTES
   2723
             2724 NXTR:
OB6E 4D05 43FA 2725 LD RETRY, $10 ! MAX TRY 10 TIMES!
OB74 OB07 OO1E 2726 CP
                             R7, #30 ! IF REMAINING BYTE(30, USE!
             2727
                                     ! (COUNT)=REMAINING BYTE!
0878 FF03
             2728
                       JR
                             NC, SET30
OB7A A179 2729 LD R9, R7 ! COUNT=REMAINING BYTE, !
OB7C 8D78 2730 CLR R7
                                    ! R7=0 INDICATE LAST!
             2731
2732
                                    ! DATA RECORD !
OB7E E804
                       JR
                             COTT
                                    ! SEND 30 BYTE FOR NOW !
! FIGURE REMAINING BYTE FOR!
0B80 2109 001E 2733 SET30: LD
                            R9, #30
            2734 SUB R7, #30 ! FIGURE REMAIN
2735 ! NEXT TIME !
0B84 0307 001E
             2736 COTT: LD
                             R11, R9
                                      ! SAVE (COUNT) INCASE NEED !
OB88 A19B
                                   ! TO SEND THE SAME RECORD!
             2737
             2738
             2739
                  ! FORMAT ADDRESS, CHECKSUM & DATA IN RECORD & !
                    ! SEND TO MCZ ONLY IF ESCAPE NOT PRESS!
             2740
             2741
OBBA 6705 43D4 2742 RESEND: BIT CFLAGS, #ESCAPE
OBBE EEEA
             2743 JR NZ, SABTS ! SEND ERROR RECORD TO MCZ!
             2744
                                    ! & MESSAGE TO USER, ABORT !
             2745
                  LD R2,#MCZPTR ! SEND DATA RECORD TO MCZ !
OB90 2102 ODBO' 2746
0B94 DF93
                      CALR RECSND
             2747
0B96 E60B
             2748
                      JR Z, LREC ! SEND LAST RECORD !
```

```
2749
              2750 CALR SKPLNE ! SKIP INPUT LINE FROM MCZ!
OR98 DF13
              2751 ! WHICH ECHO THE DAT RECORD!
              2752
OB9A DF59
              2753
                        CALR HATACK
                       JR
OB9C EE01
              2754
                              NZ, RECE79 ! EITHER RECEIVE '7' OR '9'!
OBSE ESET
              2755
                        JR
                              NXTR ! Z MEANS ECEIVE '0' !
              2756
OBAO E70D
              2757 RECE79: JR
                              CABTS
                                       I NZ,C RECEIVE '9' I
              2758
              2759
                   ! RECEIVE NON-ACKNOWLEDGE, RESEND SAME RECORD !
              2740
              2761
OBA2 83B8
              2762
                        SUB
                              R8, R11
                                       ! RECEIVED NON-ACK !
OBA4 A1B9
              2763
                        10
                              R9, R11
                                       ! BACK UP ADDR, RESTORE CNT!
OBA6 6BOO 43FA
              2764
                       DEC
                             RETRY
OBAA E60F
              2765
                        JR
                             Z, SNDART | STOP TRYING |
              2766
OBAC FREE
              2767 JR RESEND
              2768
              2769 ! SEND LAST RECORD /(ENTRY ADDR) OO (CKSUM) CR |
              2770
              2771 LREC: LD
OBAE 2102 ODBO'
                             R2,#MCZPTR
                                       ! SEND LAST RECORD TO !
              2772 CALR LASREC ! MCZ/ZDS SYSTEM!
OBB2 DF89
ORRA DE21
              2773 CALR SKPLNE ! SKIP INPUT LINE FROM MCZ!
              2774
                                       ! WHICH ECHO DATA RECORD!
              2775
              2776
              2777 CALR HATACK
OBB6 DF67
                                      ! HAIT FOR ACKNOWLEDGE !
              2778 RET Z
0BB8 9E06
                                        ! DONE ... !
              2779
OBBA EF04 2780 C2: JR
                             NC, RESD ! NZ,NC RECEIVE 7, RESEND!
OBBC 2102 OE42 2781 ABTS: LD
                             R2, #FLEMSG ! RECEIVE '9', FILE ERROR !
OBCO D187
              2782 PUTM: CALR PUTMS6
OBC2 9E08
              2783
                        RET
              2784
              2785
              2786 ! GOT '7', SEND SAME DATA IF THIS IS NOT THE 10 TIMES !
              2787
OBC4 6800 43FA
              2788 RESD: DEC RETRY
OBC8 EEEO
              2789
                        JR
                             NZ, RESEND
              2790
              2791 SNDABT:
OBCA DEE7
              2792 CALR SNDERR ! ENOUGH, STOP TRYING, SEND !
OBCC 2102 OE34
              2793
                       LD R2,#ABMSG ! 'CKSUM ERROR' TO TERMINAL!
OBDO D18F
              2794 CALR PUTMSG
OBD2 9E08
              2795 RET
OBD4
              2796 END SEND
              2797 ! PAGE EJECT!
              2798
OBD4
              2799 GLOBAL CKFNAM PROCEDURE
```

```
2801 1 *********************************
                2803 ! CKFNAM RESET TWO POINTERS OF MCZ INPUT BUFFER BEFORE
                2804 ! STARTING THE UPLOAD/DOWNLOAD PROCESS. ALSO CHECK
                2805 ! THE FILENAME IS INCLUDED IN THE COMMAND.
                2806 1
                2808
                2809 ENTRY
OBD4 4D08 43C8
                2810
                           CIR
                                   MC7PUT
ORDS 4DOS 43CA
                2811
                            CLR
                                  MCZGET ! RESET MCZ INPUT BUF POINTER!
                                   NXTARG ! SKIP OVER COMMAND !
OBDC D24B
                            CALR
                2812
                                   RLO, #'A' ! INSURE FILENAME EXIST !
OBDE 0A08 4141
                2813
                            CPR
OBE2 E703
                2814
                            JR
                                  C, WRNAME
OBE4 0A08 5B5B
                2815
                            CPB
                                   RLO, #'Z'+1
OBE8 9E07
                2816
                            RET
                                   C
                                             ! FILENAME OK !
OBEA 5E08 014E
                2817 WRNAME: JP EROR ! COMMAND SYNTAX !
                2818 END CKFNAM
OBEE
                2819
                2820
OBEE
                2821 GLOBAL LODREC PROCEDURE
                2822
                2823 ! ****************************
                2824 1
                2825 ! LODREC READ IN AN INPUT LINE INTO CONIBE FROM MCZ
                2826 ! START STOREING DATA IN CONIBF WHEN RECEIVE A '/'.
                2827 ! THE INPUT LINE IS TERMINATED BY CARRIAGE RETURN.
                2828 ! IT WILL READ IN MAXIMUM 80 ASCII CHARACTERS
                2829 ! AND IGNORE THE REST UNTIL A CARRIAGE RETURN IS
                2830 ! RECEIVED. SKIP OVER ANY CONTROL CHARACTER ( (20H )
                2831 1
                2832 1
                2833 ! OUTPUT: RETURN DATA IN CONIBF
                2834 !
                2835 ! *** RO, R2, RL3, R4, R5 ARE LOST ****
                2836 1
                2837 ! *******************************
                2838
                2839 ENTRY
                2840
                      ! WAIT FOR A '/' TO BEGIN STORING CHARACTER INTO !
                2841
                       ! CONIBF. CONIBF DOES NOT INCLUDE '/'. !
                2842
OBEF DEZA
                2843
                           CALR
                                  MCZINP
OBFO OAOB 2F2F
                2844
                           CPB
                                  RLO, #1/1
                                               ! WAIT UNTIL '/' !
OBF4 EEFC
                2845
                           JR
                                  NZ, LODREC
                2846
OBF6 2104 40B0
                2847
                           LD R4, #CONIBF
OBFA CB50
                2848
                           LDB
                                  RL3, #80
                                               ! MAX GET 80 CHARACTER!
                2849
                2850
                        ! START STORING CHARACTER, IGNORE CONTROL CHAR!
                2851
               2852
                      ! ALSO CHECK FOR CARRIAGE RETURN FOR END OF RECORD !
```

```
2853
              2854 LODRCO: CALR MCZINP
OREC DE31
              2855
OBFE 2E48
                      LDB @R4, RLO ! STORE CHAR !
              2856
              2857
OCOO OAOR ODOD
                              RLO, #ASCICR ! CHECK FOR END OF RECORD !
0C04 EE02
              2858
                        .IR
                              NZ, CKCT
0C06 DF46
              2859 CALR SKPNUL ! SKIP THE REST OF LINE!
0C08 9F08
              2860
                        RFT
              2861
              2862 CKCT:
              2863 CPB RLO, #' ' ! IGNORE CONTROL CHAR!
OCOA 0A08 2020
OCOE E7F6
              2864
                      JR C, LODRCO
OC10 A940
                             R4, #1 ! GOOD CHARACTER !
              2865
                      INC
OC12 FBOC
              2866
                        DBJNZ RL3, LODRCO ! DECREMENT CHAR COUNT!
              2867
              2868
              2869 ! ALREADY RECEIVE 80 CHARACTERS, SKIP OVER ALL !
                   ! CHARACTERS UNTIL CARRIAGE RETURN. !
              2871
OC14 DF3D
              2872 LODRC2: CALR MCZINP
                                          ! GOT 80 CHAR, IGNORE THE!
              2873
                       CPB RLO, #ASCICR ! REST OF CHAR EXCEPT CR !
OC16 OAO8 ODOD
              2874
OCIA EEFC
              2875 JR NZ, LODRC2
              2876
OC1C 6E08 4100
              2877 LDB CONIBF+80,RLO ! GOT CR !
              2878
                        RET ! WILL RETURN Z FOR OK !
0C20 9E08
              2879
              2880
0022
              2881 END LODREC
              2882
              2883
0022
              2884 GLOBAL REKCKS PROCEDURE
              2885
```

532

```
2887 1
             2888 ! REKCKS VERIEY TWO CHECKSUM VALUES IN A RECORD STORED
             2889 ! IN CONIBF. ALL ASCII CHARACTERS IN CONIBF ARE
             2890 ! CONVERTED TO 4-BIT HEX VALUE TO OBTAIN CHECKSUM WHICH!
       2891 ! IS THE SUN OF ALL THESE 4 BIT HEX VALUE.
     2892 1
       2893 ! THE FORMAT OF THE RECORD IS AS FOLLOWED:
             2894 ! (ADDRESS(4)) (COUNT(2)) (CKSUM(2)) (DATA(2))
             2895 ! (DATA(2)) ... (DATA(2)) (CKSUM(2)) (CR)
             2896 !
             2897 ! INPUT: RECORD IN CONIBF
             2898 !
             2899 ! OUTPUT: RH3 = (COUNT) IN BYTE VALUE
             2900 1
             2901! RETURN C IF CHECKSUM IS BAD
             2902 1 RETURN C TE NON-ASCIT CHAR FOUND IN CONTRE
             2903 !
                         RETURN NC IF OK
             2904 1
             2905 | **** RO, R2, R3 ARE DESTROYED ****
             2908
             2909 ENTRY
            2910 LD R2, #CONIBF
OC22 2102 40B0
             2911 LDB RH3, #3 ! 6 CHARACTERS TO CHECKSUM!
2912 CALR CHKCKS ! VERIFY 1ST CHECKSUM!
2913 RET C ! RET IF BAD!
0C26 C303
OC28 DFF9
0C2A 9E07
             2914
             2915 TESTB RH3 ! CHKCKS RETURN RH3=COUNT !
0020 8034
            2916 RET Z ! IF COUNT=0, NO DATA!
OC2E 9E06
             2917
                                     ! RET NC IF OK !
          2918 PUSH @R15, R3 ! SAVE COUNT FOR LATER!
0C30 93F3
             2919 CALR CHKCKS ! VERIFY DATA CKSUM !
OC32 DFFE
             2920
                                     ! # OF CHAR = (RH3) *2 !
            2921
OC34 97F3 2922 POP R3, OR15 ! RETREIVE COUNT IN RH3!
0C36 9E08
             2923
                      RET
            2924
          2925 END REKCKS
           2926
             2927
0C38
             2928 GLOBAL CHKCKS PROCEDURE
          2929
```

```
2030 | ******************************
               2031 1
      2932 ! CHKCKS VERIFY CHECKSUM IN THE RECORD, CONVERT ASCII
               2933 ! CHARACTER IN THE RECORD TO HEX VALUE AND ACCUMUATE
  2934 | THEIR SUM TO ORTAIN THEIR CKSUM. THEN COMPARE THE
               2935 ! CALCULATED CHSUM AGAINST THE CKSUM IN THE RECORD
               2936 ! WHICH IS FOLLOWED THE CHARACTER STRING. THE CKSUM
               2937 ! IN THE RECORD IS ALSO ASCIT CHARACTER, NEED TO
               2938 ! CONVERT TO HEX VALUE .
               2939 1
               2940 ! INPUT: R2 = POINTER TO THE ASCIT CHAR STRING
               2941 1
                             RH3 = # OF PAIR OF CHARACTERS TO CHECKSUM
                                ( TOTAL CHAR = RH3 * 2 )
               2942 1
               2943 1
               2944 ! OUTPUT: RH3 = COUNT VALUE IN THE RECORD (HEX)
               2945 ! RL3 = CALCULATED CHECKSUM (HEX)
               2946! RLO = CHECKSUM IN THE RECORD (HEX)
               2947 1
               2948 1
                             RETURN C IF CKSUM BAD
               2949! RETURN C IF FOUND NON-ASCII CHARACTER
               2950 1
                             RETURN NC IF CKSUM IS CORRECT
               2951 !
               2952 ! **** RO, R2, R3 ARE DESTROY ****
               2955
               2956 ENTRY
               2957 ! INITIALIZE CKSUM ACCUMULATOR (RL3), CONVERT!
               2958 ! THE NEXT # OF PAIR OF CHARACTERS TO HEX VALUE!
               2959 ! AND SUM UP THEIR HEX VAULE. THE HEX VALUE OF !
               2960 ! LAST PAIR OF CHARACTERS RETURN IN RLO!
               2961
OC38 8C88
               2962
                         CLRB RL3 ! INIT CKSUM ACCUMULATOR !
OC3A DFF5
               2963 CHKSM2: CALR LOBBYL ! CONVERT NEXT 2 CHAR TO !
               2964
                                            ! HEX&ADD TO CKSUM ACCUM.!
0C3C 9E07
               2965
                          RET
                                            ! FOUND NON-ASCII CHAR !
0C3E F303
               2966 DBJNZ RH3, CHKSM2 ! NEXT PAIR OF CHAR !
               2967
               2968
                       ! SAVE 8-BIT HEX VALUE OF LAST THO CHARACTERS !
               2969
                      ! (COUNT OF THE DATA IN THE RECORD). SAVE CALCULATED!
               2970
                      ! CKSUM THEN CONVERT NEXT THO CHAR TO OBTAIN CKSUM!
               2971
                      ! IN RECORD AND COMPARE THE TWO CKSUM!
               2972
0C40 A083
               2973
                          LDB
                                 RH3, RLO
                                            ! RH3=COUNT VALUE & !
0C42 93F3
               2974
                          PUSH
                                 QR15, R3 ! RL3=CALCULATED CKSUM!
               2975
OC44 DFFA
               2976
                          CALR
                                 LODBYL
                                            ! CONVERT NEXT THO CHAR TO!
               2977
                                            ! GET RECORD CKSUM IN RLO!
0C46 97F3
               2978
                          POP
                                 R3, @R15
                                            ! RETREIVE CALCULATED CKSUM!
0C48 9F07
               2979
                          RET
                                 C
                                            ! BAD CHARACTER !
               2980
OC4A 8AB8
               2981
                          CPB
                                 RLO, RL3
                                           ! COMPARE THO CKSUM !
```

```
OC4C 9E06 2982 RET Z ! RET NC TE MATCH !
           2983 SETFLG C ! RET C IF NO MATCH !
2984 RET
OC4F 8D81
0C50 9F08
              2985
0052
              2986 END CHKCKS
              2987
              2988
0052
              2989 GLOBAL LODBYL PROCEDURE
              2992 1
              2993 ! CONVERT TWO ASCII CHARACTERS TO TWO 4-BITS HEX VALUE
              2994 ! AND ADD THE THO HEX VALUES TO AN ACCUMULATOR
              2996 ! INPUT: R2 = POINTER TO THE FIRST ASCII CHARACTER
              2997 ! RL3 = ACCUMULATOR
              2998 !
              2999 ! OUTPUT: R2 = UPDATED POINTER ( INC BY 2 )
              3000 ! RL3 = UPDATED ACCUMULATOR
                         RLO = 8 BITS HEX VAULE OF THE THO ASCII CHAR !
              3001 1
              3002 1
              3003 !
                         RETURN C IF ANY CHARACTERS ARE NON-ASCII
              3004 1
                         RETURN NC IF OK
              3005 1
              3006 ! **** RO IS DESTROYED ****
              3009
              3010 ENTRY
OC52 DFF6 3011 CALR HEXDCD ! CONVERT CHAR POINTED BY !
              3012 ! R2 TO HEX IN RLO !
OC54 9E07 3013 RET C ! BAD CHARACTER!
              3014 ADDB RL3, RL0 ! ADD HEX VALUE TO ACCUM.!
3015 SLA RO, #12 ! SAVE HEX IN RHO BIT 4-7 !
0C56 808B
0C58 B309 000C 3015
OCSC DFFB 3016 CALR HEXDCD ! CONVERT NEXT CHAR!
OC5E 9E07 3017 RET C

        0C60 808B
        301B
        ADDB
        RL3, RL0
        ! ADD HEX TO ACCUMULATOR !

        0C62 840B
        3019
        ORB
        RL0, RH0
        ! MERGE THO HEX VAULE !

        0C64 8D83
        3020
        RESFLG C

0C66 9E08 3021 RET ! RETURN NC FOR OK!
              3022
8630
              3023 END LODBYL
              3024
              3025
OC68 3026 GLOBAL HEXDCD PROCEDURE
              3027
```

```
3028 | ****************************
      3029 1
             3030 ! CONVERT ASCII CHARACTER POINTED BY R2 TO CORRESPONDING!
             3031 ! 4-BIT HEX VAULE, INCREMENT POINTER R2 BY 1
             3033 ! INPUT: R2 = POINTER TO ASCII CHAR
             3034 1
             3035 ! OUTPUT: R2 = ADVANCE BY 1 & POINTED TO NEXT CHAR
                        RLO = 4-BIT HEX VALUE OF ASCII CHAR
             3037 !
             3038 1
                        RETURN C IF CHAR IS NON-ASCII
             3039 ! RETURN NC IF OK
             3040 !
             3042
             3043 ENTRY
0C68 2028
             3044 LDB RLO, @R2 ! GET ASCII CHAR !
0C6A A920
             3045 INC R2, #1 ! ADVANCE POINTER!
OC6C 5E08 O6FA' 3046 JP CONVERT ! EITHER RET C OR NC !
    3047
OC70
             3048 END HEXDCD
             3049
             3050
             3051
OC.70
             3052 GLOBAL RECSND PROCEDURE
             3054 | ******************************
             3055 !
             3056 ! RECSND FORMAT ADDRESS, COUNT, CHECKSUM & DATA IN
             3057 ! TEKTRONIX FORMAT. IT EITHER SEND THE DATA TO MCZ/ !
             3058 ! ZDS SYSTEM OR TO THE TERMINAL.
             3059 ! IF COUNT = 0 , JUST STORE '/' IN BUFFER & DON'T !
             3060 ! FORMAT THE DATA.
   3061 !
             3062 ! INPUT: R9 - COUNT, R8 - STARTING ADDRESS
             3063 ! R2 - ROUTINE ADDRESS (PUTPTR OR MCZPTR
            3064 1
       3065 ! OUTPUT: R8 INCREASE BY R9 (COUNT)
             3066 ! RETURN Z FOR LAST RECORD
             3067 !
                        RETURN NZ FOR SENDING THE RECORD & READY !
             3068 !
                               FOR ACKNOWLEDGE
             3069 !
             3070 ! **** RO, R2, R3, R5, R9 ARE LOST ****
             3073
             3074 ENTRY
OC70 8D38
             3075
                      CLR
                           R3
0C72 C82F
             3076
                      LDB
                           RLO, #1/1
0C74 6E08 4130 3077
                      LDB
                           CONOBF, RLO ! STORE '/' !
0C78 6900 43B6 3078
                      INC
                           OUTPTR, #1 ! ADVANCE CONOBF OUTPUT PTR!
             3079
```

```
3080 TEST R9 ! CK COUNT=0 FOR SENDING!
3081 RET Z ! LAST RECORD (ENTRY ADDR)!
OC7C 8094
OC7E 9E06
             3082
0C80 A185
            3083
                     LD R5, R8 ! ADD ADDRESS(4) TO CKSUM!
OC82 D1A9
             3084 CALR PUT2
                                  ! (RL3),4-BIT AT A TIME.!
             3085
                                   ! CONVERT EACH 4 BIT VALUE!
             3086
                                  ! TO ASCII CHAR AND STORE !
             3087
                                  ! THEM IN CONOBF!
OC84 A195 3088 LD R5, R9 ! STORE COUNT(2) IN CONOBF!
           3089 CALR PUT1 ! & UPDATE CKSUM (RL3) !
3090 LDB RL5, RL3
3091 CALR PUT1 ! STORE ADDR &COUNT CKSUM1!
OC86 D1A9
OC88 AOBD
OC8A DIAB
             3092
             3093 ! FORMAT (DATA) ... (DATA) (CKSUM2) !
             3094
OC8C 8D38
             3095 CLR R3 ! INIT CKSUM !
             3096 MDATA: LDB
                            RL5, QR8 ! ADD DATA TO CKSUM (RL3) !
OCBE 208D
           3097 CALR PUT1 ! 4-BIT AT A TIME. CONVERT!
OC90 DIAE
             3098 ! 4-BIT VALUE TO ASCII CHAR!
             3099
             3099 ! AND STUKE THEM IN CUMUBE:
3100 INC R8, $1 ! ADVANCE MEMORY ADDRESS!
                                      ! AND STORE THEM IN CONOBF!
0C92 A980
          3101 ! DJNZ R9, MDATA R9=COUNT OF DATA!
OC94 AB90
           3102
                    DEC R9
             3103 JR NZ, MDATA
OC96 EEFB
             3104
OC98 AOBD 3105 LDB RL5, RL3 ! STORE DATA CKSUN!
OC9A D1B3
            3106 CALR PUT1
             3107
             3108 ! SEND RECORD TO MCZ/ZDS SYSTEM!
             3109
0C9C 1F20
        3110 CALL @R2 ! CALR PUTPTR / MCZPTR !
        3111
OC9E 8D43
           3112 RESFLG Z ! RET NZ FOR NEXT REC!
OCAO 9E08
             3113
                    RET
    3114
3115 END RECSND
         3114
OCA2
             3116
            3117
OCA2
        3118 GLOBAL LASREC PROCEDURE
      3119
     3121 1
             3122 ! LASREC FORMAT ENTRY ADDRESS NITH O COUNT, CHECKSUM !
             3123 ! IN TEKTRONTIX FORMAT FOR THE LAST RECORD
             3124 ! IT EITHER SEND THE DATA TO TERMINAL OR MCZ/ZDS.
             3125 !
             3126 ! INPUT : R6 - ENTRY ADDRESS
             3127 ! R2 - ROUTINE ADDRESS (PUTPTR OR MCZPTR)
             3128 !
             3129 ! **** RO, R2, R3, R5 ARE LOST ****
             3130 1
```

```
3132
3133 ENTRY
          3134 LD R5, R6
OCA4 D1BA 3135 CALR PUT2
OCA6 CDOO 3136 LDB
                        RL5, #0 ! 00 IN CONOBF !
OCAS DIBA 3137 CALR PUT1
OCAA AOBD 3138 LDB
                        RL5, RL3 ! CKSUM !
OCAC DIBC 3139 CALR PUT1
OCAE 1E28 3140 JP
                         @R2 ! SEND TO TERMINAL OR MCZ!
        3141
OCBO
           3142 END LASREC
         3143
           3144
OCBO
           3145 GLOBAL STREC PROCEDURE
           3147 ! наказания наказания выправания выправания выправания в
           3148 !
           3149 ! STREC UNPACKS DATA IN RECORD & STORE THEM IN HEMORY !
     3150 ! LOCATION WHICH SPECIFIED IN THE RECORD.
  3151 !
   3152 ! INPUT: RECORD IN CONIBF
       3153 ! RH3 - # OF BYTES OF DATA IN CURRENT RECORD
           3154 !
           3155 ! **** RO, R1, R2, R3 R4 ARE LOST ****
           3158 ENTRY
           3159
OCBO AO3C
           3160 LDB RL4, RH3 ! SAVE COUNT
           3161
           3162 ! CONVERT (ADDRESS) & (DATA) TO HEX VALUE.STORE!
           3163 ! BYTE DATA IN MEMORY. THE # OF BYTE OF DATA IS !
           3164 ! SPECIFIED BY (COUNT) !
           3165
           3166 CALR RECADR ! GET RECORD ADDR IN R1 ! 3167 LD R2, #CONIBF+8 !ADDRESS OF 1ST DATA !
OCB2 DFF9
OCB4 2102 40B8 3167
           3168
OCB8 D034
           3169 LODFR3: CALR LODBYL
                               ! CONVERT NEXT 2 CHAR TO !
           3170 ! BYTE DATA IN RLO !
           3171 LDB @R1, RLO ! STORE DATA IN MEMORY !
OCBA 2E18
OCBC A910
         3172 INC R1, #1 ! ADVANCE MEMORY ADDRESS !
OCBE FC04
           3173 DBJNZ RL4, LODFR3
OCCO 9E08 3174 RET
           3175
       3176 END STREC
           3177
           3178
OCC2
      3179 GLOBAL RECADR PROCEDURE
           3180
```

```
3182 1
              3183 ! CONVERT THE STARTING ADDRESS IN LOAD RECORD TO HEX
              3184 ! VALUE IN R1
              3185 1
              3186 ! никиминичения виничения виничения виничения виничения
              3187
              3188 ENTRY
OCC2 2102 40B0 3189 LD R2, #CONIBF ! CONVERT NEXT 4 ASCII CHAR! OCC6 DO3B 3190 CALR LODBYL ! TO HEX ADDRESS IN R1! OCCB A0B1 3191 LDB RH1, RLO ! 1ST BYTE OF AN ADDRESS!
OCCA DO3D 3192 CALR LODBYL
OCCC A089 3193 LDB RL1, RLO ! 2ND BYTE OF ADDRESS !
OCCE 9E08 3194 RET
0000
              3195 END RECADR
              3196
              3197
         3198 GLOBAL GODCKS LABEL
OCDO
              3199 GLOBAL BADCKS PROCEDURE
              3200
              3201 | *****************************
      3202 1
              3203 ! BADCKS SEND NON-ACKNOWLEDGE (ASCII 7 (37H) & CR ) FOR !
              3204 ! RECEIVING NON-ASCII CHARACTER OR CHECKSUM IN THE !
              3205 | RECORD IS INCORRECT
              3206 1
              3207 ! GODCKS SEND ACKNOWLEDGE (ASCII 0 (30H) & CR) FOR
              3208 ! RECEIVING A VALID RECORD OF DATA
              3209 1
              3210 ! ABKCKS SEND ABORT-ACKNOWLEDGE (ASCII 9) FOR USER
              3211 ! WANT TO ABORT THE DOWNLOAD PROCESS. !
              3212 1
              3213 ! **** RO, R1, R2 IS DESTROYED ****
              3214 !
              3216
              3217 ENTRY
             3217 ENTRY
3218 LDB RLO, $'7'
OCDO C837
              3219 JR MSEND
OCD2 E803
              3220
OCD4 C839
              3221 ABKCKS: LDB RLO, #'9'
              3222 JR MSEND
OCD6 E801
              3223
OCD8 C830 3224 GODCKS: LDB RL0,#'0'
OCDA 6E08 4130
             3225 MSEND: LDB CONOBF, RLO
OCDE 4D05 43B6 3226 LD
                              OUTPTR,#CONOBF+1
OCE2 4131
OCE4 DF9B
              3227 CALR MCZPTR
                                       ! OUTPUT CONBOF!
OCE6 DFBA
              3228
                        CALR
                            SKPLNE ! SKIP INPUT LINE FROM MCZ!
              3229
                                         ! WHICH ECHO ACKNOWLEDGE!
OCE8 9E08
              3230
                     RET
              3231
```

```
3232 END BADCKS
OCFA
              3233
              3234
OCEA
              3235 GLOBAL WATACK PROCEDURE
              3236
              3238 1
              3239 | EXPECT EITHER ACKNOWLEDGE (ASCII 0) OR NON-ACKNOWLEDGE!
              3240 ! (ASCII 7) FROM SERIAL LINE.
              3241 | IGNORE OTHER CHARACTERS
              3242 1
                          RETURN Z, NC RECEIVE GOOD-ACK FROM MCZ
              3243 1
              3244 ! RETURN NZ, NC RECEIVE NON-ACK FROM MCZ
                          RETURN NZ,C RECEIVE ABORT-ACK FROM MCZ
              3245 1
              3246 !
              3247 ! **** RO IS DESTROYED ****
              3248 1
              3250
              3251 ENTRY
              3252 CALR MCHAR ! GET CHAR!
OCEA DEZE
              3253 CPB RL1, *'0' ! CHECK FOR ACKNOWLEDGE!
3254 JR NZ, CK79 ! NO, CHECK FOR 7 OR 9 !
3255 CALR SKPLNE ! YES, SKIP THE LINE!
OCEC 0A09 3030
OCFO EE04
OCF2 DFC0
                      SETFLG 7
OCF4 8D41
              3256
OCF6 8D83
              3257
                       RESFLG C
            3258 RET
OCF8 9E08
                                      ! RETURN Z,NC FOR GOOD-ACK !
              3259
OCFA 0A09 3737 3260 CK79: CPB RL1, *'7' ! CHECK FOR NON-ACK. ! OCFE EE04 3261 JR NZ, CK9 ! NO!
              3262 CALR SKPLNE ! SKIP RESPONSE LINE!
ODOO DFC7
ODO2 8D43
              3263 RESFLG Z
              3264 RESFLG C
ODO4 8D83
              3265 RET 3266
0D06 9E08
                                         ! RETURN NZ,NC FOR NON-ACK !
ODO8 0A09 3939 3267 CK9: CPB RL1, #'9'
              3268 JR Z, MATC
ODOC E602
                                        ! YES, ABORT-ACK !
                       CALR MCZINP
              3269
                                      ! ADVANCE POINTER TO GET!
ODOE DEBA
              3270
3271 JR WATACK
                                       ! NEXT CHAR !
ODIO ESEC
         3272 MATC: CALR SKPLNE
3273 RESFLG Z
                                       ! SKIP OVER RESPONSE LINE!
OD12 DFDO
OD14 8D43
OD16 8D81
             3274
                       SETFLG C
OD18 9E08
              3275
                       RET
                                       ! RETURN NZ,C FOR ABORT-ACK!
              3276 END WATACK
OD1A
              3277
              3278
              3279
ODIA 3280 GLOBAL SNDCMD PROCEDURE
    3281
```

```
3283 !
              3284 ! SNDCMD SEND THE RIO COMMAND 'B;' FOLLOWED BY THE !
              3285 ! COMMAND RECEIVED FROM THE TERMINAL TO THE MCZ. !
              3286! THE 'B;' FORCE MCZ TO BE IN BRIEF MODE.
              3287 ! THEN SKIP OVER THE INPUT LINE RECIEVED FROM MCZ THAT!
              3288 ! ECHO BACK THE COMMAND JUST SEND. IT THE MCZ IS IN !
              3289! VERBOSE MODE, SKIP OVER THO LINES. THEN CHECK THE!
              3290! FIRST CHARACTER IN THE FOLLOWING INPUT LINE RECEIVED!
              3291 ! FROM MCZ. IF THE CHARACTER IS NOT AN VALID
              3292 ! ACKNOWLEDGE ( ASCII 0, 9, 7 ), THEN PRINT THE
              3293 ! INPUT LINE TO THE TERMINAL, THIS MEANS THE Z80
              3294 ! PROGRAM DID NOT GET LOADED.
              3295 !
              3296 ! OUTPUT: RETURN NZ IF MCZ GET ZBO PROGRAM LOADED.
              3297 ! RETURN Z IF MCZ DID NOT LOAD THE Z80 PROGRAM !
              3300
              3301 ENTRY
OD1A 6705 43D4 3302 BIT CFLAGS, #ESCAPE
              3303 JR Z, NOESC 383304 SETFLG Z
OD1E E602
OD20 8D41
              3305
3306
0D22 9E08
                      RET
                                          ! RETURN Z FOR PROGRAM DID!
                             ! NOT LOADED !
              3306
              3307 NOESC:
OD24 2101 423B 3308 LD R1, #X423B ! SEND 'B;' TO RIO TO FORCE!
0D28 6F01 4130
              3309
                      LD CONOBF,R1 ! IN BRIEF MODE!
              3310
OD2C 2100 4132
              3311 LD RO, #CONOBF+2
OD30 2101 40B0
              3312 LD R1, #CONIBF ! TRANSFER COMMAND TO !
OD34 2102 0040
              3313 LD R2, $%40 ! CONOBF BUFFER!
              3314 LDIR 3315 LD
OD38 BB11 0200
                              eRO, eR1, R2
OD3C 4D05 43B6
                              OUTPTR,#CONOBF+%80
OD40 41B0
OB42 DFC4
              3316 CALR MCZNCR
                                          ! SEND CONOBF TO MCZ !
              3317
3318 ! SKIP OVER MCZ INPUT LINE THAT ECHO THE COMMAND !
              3319
OB44 DFE9
              3320 CALR SKPLNE
                                       ! SKIP OVER SINGLE LINE !
              3321
OD46 DFAD 3322 CALR MCHAR ! MAIT FOR RESPONSE FROM MCZ!
              3323
                                       ! EITHER ECHO 'B' OR RESPONSE!
              3324
                         FROM Z80 PRGRAM OR RESPONSE!
                                     ! FROM RIO !
              3325
              3326
             3327
OD48 0A09 4242
                      CPB RL1, #'B' ! IF MCZ IN VERBOSE MODE, IT !
004C FF02
              3328 JR NZ, CKRIO! HILL SEND AN EXTRA LINE!
             3329 CALR SKPLNE ! SKIP ANOTHER LINE ! 3330 CALR MCHAR ! WAIT FOR EVER !
OD4E DFEE
                       CALR MCHAR
OBSO DFB2
              3330
                                       ! WAIT FOR EVER !
              3331 CKRIO:
OD52 0A09 3939 3332 CPB RL1,#'9' ! CK IF PROGRAM GET LOADED !
```

```
! OR MCZ STILL IN RIO !
            3333
            3334 JR Z, OKRET ! RETURN NZ FOR SOME ACKNOWL.!
0D56 E60C
OD58 0A09 3030 3335 CPB RL1, #'0'
             3336 JR Z, OKRET
0D5C E609
            3337 CPB RL1, #'7'
ODSE 0A09 3737
            3338 JR Z, OKRET
0D62 E606
            3339
            3340 ! ECHO INPUT LINE RECEIVED FROM MCZ TO TERMINAL!
            3342 OUTCH:
            3343 CALR MCZINP ! ECHO THE MCZ RESPONSE TO !
OD64 DFE5
                           ! TERMINAL !
            3344
            3345 KEP: (384)
OD66 D2F9 3346 CALR TYWR ! TERMINAL WHY Z80 PRGRAM!
OD68 OAO8 OAOA 3347 CPB RLO, #ASCILF! DIDNOT GET LOADED. !
            3348 RET Z ! BACK TO DEBUG!
OD6C 9E06
ODSE EBFA
            3349
                 JR OUTCH
            3350
            3351
OD70 8D43
0D70 8D43
0D72 9E08
            3352 OKRET: RESFLG Z
                                   ! RETURN NZ FOR RECEIVE !
            3353 RET ! SOME ACKNOWLEDGE!
0D74
            3354 END SNDCHD
            3355
            3356
0D74
            3357 GLOBAL SKPLNE PROCEDURE
            3360 1
            3361 ! SKPLNE SKIP OVER A INPUT LINE RECEIVED FROM MCZ.
             3362 ! INCLUDING ALL THE NULL CHARACTER AT THE END OF LINE !
            3363 ! THE FIRST CHARACTER OF THE NEXT LINE WILL RETURN !
            3364 ! IN RL1, AND THE POINTER (MCZGET) STILL POINT TO
            3365 ! THE FIRST CHARACTER.
            3366 !
             3367 ! OUTPUT: RL1 - FIRST CHARACTER OF NEXT MCZ INPUT LINE!
             3368 ! RETURN Z IF ESCAPE KEY NOT HIT
             3369 1
                        RETURN NZ USER PRESS ESCAPE KEY
            3370 !
             3372
            3373 ENTRY
            3374 CALR MCZINP ! SKIP OVER INPUT LINE!
OD74 DFED
            3375
                                 ! UNTIL A LF !
            3376 CPB RLO, #ASCICR ! UNTIL A CR !
OB76 OAO8 OBOD
             3377 JR NZ, SKPLNE
OD7A EEFC
             3378
            3379
            3380 ! SKIP OVER LINE FEED & NULL CHARACTER !
            3381 ! HAIT FOR A WHILE FOR THE NEXT CHARACTER TO COME IN !
            3382
            3383
OD7C 2101 3000 3384 SKPNUL: LD R1,#X3000 ! CONSTANT LOOP!
```

```
0D80 6100 43CA 3386 LD RO,MCZGET
 OD84 4B00 43C8 3387
                         CP RO,MCZPUT
 OD88 EE03 3388 JR NZ,GOTCHS
OD8A AB10 3389 DEC R1,≇1 ! WAIT ...!
 ODSC EEF9 3390 JR NZ,XX
 ODRE 9FOA
                3391 RET Z ! END OF LINE !
                 3392
                 3393 GOTCHS:
3394 CALR MCHAR ! GET CHAR, BUT DON'T ADVANCE!
 0D90 DFD2
                 3395 ! POINTER!
                 3396
3396
OD92 OAO9 2020 3397 CPB RL1,*'' ! LF & NULL ( SPACE !
                3398 RET PL ! GOT ASCII CHAR !
3399 CALR MCZINP ! SKIP THE LF OR NULL !
OD96 9EOD
OD98 DFFF
OD9A E8FO
                           JR SKPNUL
                3400
                3401
                 3402 END SKPLNE
 ONSC
                 3403
                 3404
ODEC
                 3405 GLOBAL MCZINP PROCEDURE
                 3406
                 3408 1
                 3409 ! MCZINP GET A CHARACTER FROM THE MCZ INPUT INTERRUPT !
                 3410 ! BUFFER. IF NO CHARACTER IN THE BUFFER, IT WAIT....!
                 3411 1
                 3412 !
                              MCZPUT UPDATED TO POINT TO NEXT CHAR
                 3413 !
                             CHARACTER RETURN IN RL1
                 3414 1
                 3415 ! #### RO, R2 ARF LOST #####
                 3418
                 3419 ENTRY
OD9C 6100 43CA 3420 LD RO, MCZGET
ODAO 4B00 43CB 3421 CP RO, MCZPUT ! CHECK ANY CHAR !
ODA4 E6FB 3422 JR Z, MCZINP ! NO, WAIT....!
                3423

        ODA6
        D2DC
        3424
        CALR
        MCZADR
        ! GET ADDRESS

        ODA8
        6F00
        43CA
        3425
        LD
        MCZGET, RO

        ODAC
        2028
        3426
        LDB
        RLO, @R2
        ! GET CHAR !

                                           ! GET ADDRESS OF BUF !
ODAC 2028
ODAE 9E08
                3427 RET
                                   ! WILL RETURN NZ !
                3428
ODBO
                3429 END MCZINP
                3430
                3431
ODBO
                3432 GLOBAL MCZPTR PROCEDURE
                3433
```

```
7475 1
                                    3436 ! MCZPTR OUTPUT CHARACTER STREAM IN CONOBE TO MCZ/ZDS
                                    3437 ! SYSTEM. OUTPIR IS THE CONOBE ADDRESS CONTAINING
                                    3438 | THE LAST CHARACTER.
                                    3439 ! THE LAST CHARACTER. IT ALSO ADD A CARRIAGE RETURN !
                                    3440 ! TO THE END OF THE CHARACTER STREAM.
                                    3442 | INPUT: DATA IN CONOBE, OUTPIR
                                     3443 !
                                     3444 ! OUTPUT: OUTPTR = #CONOBF, BLANK IN CONOBF BUFFER
                                     3445 1
                                     3447
                                     3448 ENTRY
                                    3449 LD R2, OUTPTR
3450 LDB @R2, #ASCICR ! ADD CARRIAGE RETURN !
3451 INC OUTPTR, #1
3452
ODBO 6102 43B6
ODB4 OC25 ODOD
ODBS 6900 4386
                                     3452
                                     3453 MCZNCR:
ODBC 2101 4130 3454 LD R1, #CONOBF
                                     3455 LOPS: LDB RLO, @R1
ODCO 2018
                      3456 INC R1
ODC2 A910
                                     3457
                                    345B CALR MCZSND ! SEND DATA TO MCZ !
ODC4 D31C
ODC6 E603 3459 JR Z, ENDS
                                     3460
                                     3461
ODCB 4B01 43B6 3462 CP R1, OUTPTR ! CHECK IF ALL DONE!
                                     3463 JR C, LOPS
                                                                                                                ! NOT YET !
ODCC E7F9
ODCE 4D05 43B6 3465 ENDS: LD OUTPTR, #CONOBF
 ODD2 4130
ODD4 2102 0040 3466 LD R2, #COSIZE/2
ODD8 AB20 3467 DEC R2,#1
ODDA 4D05 4130 3468 LD CONOBF, #' '
ODDE 2020
 ODDE 2020
 ODEO 2100 4130 3469 LD RO, #CONOBF
ODE4 2101 4132 3470 LD R1, #CONOBF+2
ODEB BBO1 0210 3471 LDIR @R1, @R0, R2 ! BLANK BUFFER !
 ODEC 9E08
                                     3472
                                                           RET
                                      3473
                                     3474 END MCZPTR
 ODEE
                                      3475
                                     3477 GLOBAL MCHAR PROCEDURE
 ODEF
                                     3478 ENTRY
| 3479 | 3479 | 3479 | 3479 | 3479 | 3479 | 3479 | 3479 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 | 3470 |
```

```
ODFA 2029
                3484 LDB RL1, eR2 3485
 ODFC 9E08
                                        ! WILL RETURN Z FROM !
                 3486
                 3487
                                             ! MCZADR ROUTINE !
 ODEF
                 3488 END MCHAR
                 3489
                3490
 ODEF
                3491 GLOBAL SNDERR PROCEDURE
                3493 ! SEND ASCII '//' TO MCZ TO ABORT THE Z80 SEND PRGRAM !
                3494
                3495 ENTRY
                3496 LD RO, #'//'
3497 LD CONOBF,RO
3498 LD OUTPTR,#CONOBF+2
 ODFE 2100 2F2F
 0E02 6F00 4130
 0E06 4D05 43B6
 OEOA 4132
                3499 CALR MCZPTR 3500 RET
 OEOC DOOF
 0E0E 9E08
 0F10
                3501 END SNDERR
                3502 GRAD 3502
 0F10
                3504 GLOBAL MSG PROCEDURE
                3505 ENTRY
                3506 BARAT:
0E10 09
              3507
                           BUAL 9
0E11 42
              3508
                           BUAL 'B'
        3509
3510
3511
3512
0E12 41
                           BVAL
                               141
0E13 44
                                 101
                           BVAL
0E14 20
                           RUAL
                                 1 1
0E15 44
                           BVAL
                                 101
0E16 41
           3513
                           BVAL
                                 141
0E17 54
             3514
                           BVAL
                                 171
0E18 41
             3515
                           BVAL
                                 IAI
             3516
0E19 OD
                           BVAL
                                 ZOD
               3517
0E1A 19
               3518 ADBMSG:BVAL
                                 25
0E1B 2F49
               3519
                         HUAI
                               1/11
0E1B 4E43
               3520
                         HVAL
                               PNC
                         NVAL 'OR'
0E1F 4F52
               3521
0E21 5245
               3522
                              'CT'
' L'
'OA'
' D '
'AD'
0E23 4354
               3523
                         HVAL
0E25 204C
               3524
                         WVAL
0E27 4F41
               3525
                         HVAL
0E29 4420
               3526
                         HUAI
0E2B 4144
               3527
                         WVAL
0E2D 4452
               3528
                         HVAL
                              'DR'
0E2F 4553
                                'ES'
               3529
                         HVAL
0E31 5320
               3530
                         WVAL 'S'
OE33 OD
               3531
                         BVAL
                                 ZOD
               3532
0E34 OD
               3533 ABMSG: BVAL
                                13
0E35 2F43
               3534
                         WVAL
                               1/01
```

| 0E77 | 4B53 | 3535 N | IVAL | 'KS' | | |
|------|---------|----------------------|-------|-----------|--|--|
| | 554D | | IVAL | , nu, | | |
| | | | IVAL | 1 E1 | | |
| | 5252 | | IVAL | 'RR' | | |
| | 4F52 | | IVAL | | | |
| 0E41 | | | BVAL | ZOD | | |
| VLIA | | 3541 | | | | |
| 0E42 | 13 | 3542 FLENSG: | RUAL | 19 | | |
| | 2F46 | | WAL | 1/F1 | | |
| VEA | 7404 | 3544 | | | | |
| | 4520 | | HVAL | 1E 1 | | |
| | 5752 | | HVAL | 'WR' | | |
| | 3 4954 | | HVAL | | | |
| | 4520 | | | | | |
| | 4552 | 3549 | | 'ER' | | |
| | 524F | | HVAL | 'RO' | | |
| | 3 5220 | | | | | |
| | 5 OD | | BVAL | | | |
| VCJ. | 1 40 | 3553 | DAUC | WAN | | |
| ACE | / 44 | 3554 OPNMSG: | DILAI | 17 | | |
| | 6 11 | | WVAL | 1/01 | | |
| | 7 2F4F | | WVAL | | | |
| | 9 5045 | | | 'N' | | |
| | B 4E20 | | WVAL | 'FI' | | |
| | D 4649 | | HVAL | | | |
| | F 4C45 | | WVAL | | | |
| | 1 2045 | | WVAL | | | |
| | 3 5252 | | HVAL | | | |
| | 5 4F52 | | WVAL | | | |
| 0£6 | 7 OB | | BVAL | %OD | | |
| AP. | n 43 | 3564 75/5 FERRICO | DILAI | | | |
| | 8 07 | 3565 ESCMSG: | | 7 '/A' | | |
| | 9 2F41 | | WVAL | | | |
| | B 424F | | WVAL | 'BO' | | |
| | D 5254 | | HVAL | 'RT' | | |
| 0E6 | F OD | | BVAL | XOD | | |
| | | 3570 | | 1511 | | |
| | 0 454E | | HVAL | 'EN' | | |
| | 2 5452 | 3572 | WVAL | 'TR' | | |
| | 4 5920 | 3573 | WVAL | 17 1 | | |
| | 6 504F | 3574 | WVAL | 'PO' | | |
| | 8 494E | 3575 | WVAL | 'IN' | | |
| | A 5420 | 3576 | WVAL | 11, | | |
| 0E | T. | 3577 END MS | נו | | | |
| | | 3578 | | | | |
| | | 3579 | MEGGW | poorroune | | |
| 0E | /C | 3580 GLOBAL | MESSX | PRUCEDURE | | |
| | | 3581 ENTRY | | 10.1 | | |
| | 7C 3020 | 3582 LRO: | WVAL | 10 1 | | |
| | 7E 3120 | 3583 | HVAL | 11 1 | | |
| 08 | 30 3220 | 3584 | WVAL | 12 1 | | |
| 0E | B2 3320 | 3585 | HVAL | 13 1 | | |
| 0E | B4 3420 | 3586 LR4: | HVAL | 14 1 | | |

```
0E86 3520
            3587 WVAL '5 '
0E88 3620 3588 NVAL '6'
0F8A 3720 3589
                          HUAL 17 1
0F8C 3820 3590 1R8:
                          WVAL 18 1
0E8E 3920 3591
                               191
                          NVAL '10'
NVAL '11'
                          HUAL
0E90 3130
               3592
         3593
0E92 3131
0E94 3132 3594 LR12: WVAL '12' 0F94 3133 3595 WVAL '13'
0E98 3134
         3596 LR14: WVAL '14'
0E9A 3135 3597 LR15: WVAL '15'
               3598 LPC: WVAL 'PC'
0E9C 5043
0E9E 4643 3599 LFC: WVAL 'FC'
0EA0 524C 3600 LIT: WVAL 'RL'
0EA2 5248 3601 WVAL 'RH'
0EA2 5248 3601 WVAL 'RH'
0EA4 09 3602 BRKMSG: BVAL X09
0EA5 42 3603 BVAL 'B'
0EA6 5245 3604 HVAL 'RE'
HVAL 'T'
0EAE 05 3608 NHINSG: BVAL 205
0EAF 4E 3609 BVAL 'N'
0EB0 4D49 3610 MVAL 'MI'
0EB2 20 3611 BVAL '.'
0EB3 0B 3612 BUAL
                                 ZOD
OEB4 3613 END MESSX
               3614
0EB4 3616 GLOBAL INTER PROCEDURE
3617 ENTRY
             3618 RAMBTA:
OEB4 40B0 3619 HVAL CONIBF ! INPTR !
0EB6 4130 3620 HVAL CONOBF ! QUTPTR ! 0EBB 0B 3621 BVAL BACKSP ! CHRDEL !
OEBB 08 3621 BVAL BACKSP ! CHKDEL !

OEBP 7F 3622 BVAL DELTE ! LINDEL !

OEBA 11 3623 BVAL XONC ! XONCHR !

OEBB 13 3624 BVAL XOFFC ! XOFCHR !

OEBC 0000 3625 HVAL XOFOO ! NULLCT !

OEBC 0EO0 3626 HVAL XOE00 ! ILEGAL INSTURCTION!

OECO 0004 3627 HVAL XOO04 ! COUNT FOR CTC3 !

OECO 0001 3628 HVAL XOO01 ! N_ COUNT FOR BREAK & NEXT !
                                        ! COUNT FOR CTC3 !
           3629
             3630
 3631
       3632 INIOUT: BVAL %(2)00011000 ! BIT 3,4,5 CHANNEL RESET!
0EC4 18
OEC5 02 ! WRITE REG 2 !
OEC6 10 3634 BVAL X10 ! INTERRUPT VECTOR 10 !
 3635
       3636
                          BVAL
                                %(2)00010100 ! BIT 3-5 RESET EXTERNAL/!
    3637
                         ODITION TO STATUS INTERRUPT ! TO SEE
   3638
                                             ! BIT 0,1,2 WRITE REG 4 !
```

| 4500 80 | 3639 | Butt | V 20 44004453 | | 0030 4830 |
|-----------|--------------|-------|-----------------|---|---|
| OEC8 CC | 3640 | BVAL | χ(2)11001100 | | BIT 0,1 NO PARITY! |
| | 3641 | | | | BIT 2,3 TWO STOP BITS! BIT 4,5 EIGHT BITS SYNC! |
| | 3642 | | | | BIT 6,7 CLOCK RATE X 64! |
| | 3643 3644 | | | ! | BIT OF LLUCK RATE X 04: |
| OEC9 03 | 3645 | BVAL | 3 | 1 | WRITE REG 3 ! |
| OECA CI | 3646 | BVAL | %(2)11000001 | | BIT O RECEIVE ENABLE ! |
| VLGN GI | 3647 | DANC | 7/2/11/00001 | - | BIT 1 NO SYNC CHAR LOAD! |
| | 3648 | | | i | INHIBIT! |
| | 3649 | | | • | BIT 2 NO ADDRESS SEARCH ! |
| | 3650 | | | i | MODE! |
| | 3651 | | | 1 | BIT 3 NO CRC ENABLE ! |
| | 3652 | | | | BIT 4 NO HUNT MODE ! |
| | 3653 | | | | BIT 5 NO AUTO ENABLE ! |
| | 3654 | | | | BIT 6,7 EIGHT BITS PER ! |
| | 3655 | | | ! | |
| | 3656 | | | | |
| OECB 05 | 3657 | BVAL | 5 | ! | WRITE REG 5 ! |
| OECC EA | 3658 | BVAL | %(2)11101010 | ! | BIT O X'MIT CRC DISABLE! |
| | 3659 | | | ! | BIT 1 RTS ACTIVE ! |
| | 3660 | | | ! | BIT 2 SDLC/CRC16 DISABLE! |
| | 3661 | | | ! | BIT 3 X'MIT ENABLE ! |
| | 3662 | | | | BIT 4 NO SEND BREAK ! |
| | 3663 | | | ! | BIT 5,6 EIGHT BIT/CHAR! |
| | 3664 | | | ! | ON X'HIT! |
| | 3665 | | | 1 | BIT 7 DTR ACTIVE ! |
| 4500 44 | 3666 | 81141 | W/20 00040004 | | DIT TE DESET EVICENCE (I |
| OECD 11 | | BVAL | %(2)00010001 | | BIT 3-5 RESET EXTERNAL! |
| | 3668 | | | ! | STATUS INTERRUPT ! BIT 0,1,2 WRITE REG 1 ! |
| | 3669 3670 | | | : | BIT UTITZ WRITE REG I : |
| OECE 1C | | BVAL | 7(2)00011100 | 1 | BIT O EXTERNAL ! |
| ALOE ID | 3672 | Aur | WITLAAATTAA | i | INTERRUPT OFF ! |
| | 3673 | | | i | BIT 1 X'MIT INTERR.OFF! |
| | 3674 | | | i | BIT2 STATUS ASFFECT! |
| | 3675 | | | i | VECTOR ! |
| | 3676 | | | | BIT 3,4 RECEIVE ! |
| | 3677 | | | ! | INTERRUPT ENABLE ! |
| | 3678 | | | i | BIT X WAIT READY DISABLE! |
| OECF OO | 3679 | BVAL | 0 | | |
| I THERE I | 3680 | | | | |
| | 3681 | | | | |
| | 3682 | | | | |
| 0ED0 18 | | BVAL | %(2)00011000 | ! | BIT 3,4,5 CHANNEL RESET ! |
| | 3684 | | | | |
| 0ED1 14 | 3685 | BVAL | 2(2)00010100 | | BIT 3-5 RESET EXTERNAL/! |
| | | | | | STATUS INTERRUPT! |
| | 3687 | | | ! | BIT 0,1,2 WRITE REG 4 ! |
| AFRA DE | 3688 | | W/83 44 84 44 A | | DIT A 4 NO OFFITH I |
| OED2 CC | 3689 | BVAL | %(2)11001100 | | BIT 0,1 NO PARITY! |
| | 3690 | | | 1 | BIT 2,3 TWO STOP BITS! |

```
! BIT 4,5 EIGHT BITS SYNC!
            3691
       3692
                        ! BIT 6,7 CLOCK RATE X 64!
       3693
                 BVAL
                          3 ! WRITE REG 3!
      3694
0ED3 03
OED4 C1
            3695
                    BUAL
                          %(2)11000001 ! BIT O RECEIVE ENABLE !
            3696
                                ! BIT 1 NO SYNC CHAR LOAD!
                                 ! INHIBIT!
            3697
            3498
                                ! BIT 2 NO ADDRESS SEARCH !
            3699
                                ! MODE !
                                ! BIT 3 NO CRC ENABLE !
            3700
            3701
                                ! BIT 4 NO HUNT MODE !
                                ! BIT 5 NO AUTO ENABLE !
            3702
                                ! BIT 6,7 EIGHT BITS PER !
            3703
                                ! CHAR ON RECIVED !
            3704
            3705
0ED5 05
            3706 BVAL
                          5 ! NRITE REG 5 !
            3707 BVAL
                          X(2)11101010 ! BIT O X'MIT CRC DISABLE!
OED6 EA
            3708 A(2)1110
                               ! BIT 1 RTS ACTIVE !
                                ! BIT 2 SDLC/CRC16 DISABLE!
            3710
                                ! BIT 3 X'HIT ENABLE !
            3711
                                ! BIT 4 NO SEND BREAK !
            3712
                                ! BIT 5,6 EIGHT BIT/CHAR!
            3713
                                   ! ON X'MIT!
                                ! BIT 7 DTR ACTIVE !
            3714
            3715
                BVAL
                                 ! BIT 3-5 RESET EXTERNAL/!
0ED7 11
            3716
                        %(2)00010001
                         ! STATUS INTERRUPT !
            3717
    3718
                                ! BIT 0,1,2 WRITE REG 1 !
            3719
                    BVAL
OED8 18
            3720
                        %(2)00011000 ! BIT O EXTERNAL !
            3721
                 INTERRUPT OFF ! SOME ASS
            3722
                                ! BIT 1 X'MIT INTERR.OFF!
            3723
                                ! BIT2 NO STATUS ASFFECT!
                                ! VECTOR !
            3724
                                ! BIT 3,4 RECEIVE !
            3725
            3726
                                ! INTERRUPT ENABLE !
            3727
                                BIT X WAIT READY DISABLE!
            3728 BVAL
0ED9 00
            3729
            3730
            3731
            3732
            3733
0EDA 0000
            3734 PSBLK: NVAL 0
                               ! RESERVED !
OEDC 0000
           3735 WUAL
                         0
0EDE 4000
           3736
                    WVAL %4000
                               ! NON-IMPLEMENTED INSTRUCT !
0EE0 0000
           3737
                    WVAL
                         0
                               ! BREAK INTERRUPT ADDR !
0EE2 0000
           3738
                    WVAL
                        0
                               ! PREVILEGED INSTRUCTIONS !
0EE4 0000
           3739
                    HVAL
                        0
0EE6 0000
           3740
                    WVAL
                         0
                               ! SYSTEM CALL !
0EE8 0000
           3741
                    HVAL
                         0
OEEA 0000
           3742
                    WVAL
                         0
                              ! SEGMENTATION TRAP !
```

```
OFFC 0000 3743 HVAL
                            0
0EEE 4000 3744 WVAL
                            24000
                                    ! NON-MASKABLE INTERRUPT!
                                     I RETURN TO MONITOR MODE!
OFFO 0000
             3745
                       MUAI
                            0
                                     ! NON-VECTORED INTERRUPT !
OFF2 0000
           3746
                       WVAL
        3747
                            0
0EF4 0000
                       WVAL
                                     ! VECTORED INTERRUPT - FCM !
0EF6 4000
           3748
                       WUAL
                            24000
             3749
                                    ! VECTOR O!
OFFR 0000
                       MUAI
                            20000
           3750 WVAL
                            20000
                                     I VECTOR 1 !
0EFA 0000
                                    ! VECTOR 2 !
             3751 HVAL
                            20000
OEFC 0000
       3752
                                     I VECTOR 3 I
OFFF 0000
                       HUAI
                            0
                                    1 VECTOR 4 1
0F00 0000
       3753
                       WUAL
                                    ! VECTOR 5 !
0F02 0000
        3754
                       HVAL
                            0
                                    ! VECTOR 6 CTC3 BREAK & NEXT!
0F04 0000
       3755
                       HUAL
                                    I VECTOR 7 I
       3756
                       WVAL
0F06 0000
          3757
                       HVAL
                                    ! VECTOR 8 !
0F08 0000
          3758
                       HVAL
                            0
                                    I VECTOR 9 !
0F0A 0000
OFOC 0000
         3759 WVAL
                            0
                                    ! VECTOR A !
                                     ! VECTOR B !
         3760 WVAL
                            0
OFOE 0000
                                    ! VECTOR OC !
OF10 0000
        3761 WVAL
                             0
0F12 0000
       3762 WVAL
                            0
                                    ! VECTOR OD !
                                    ! VECTOR OE !
        3763 WVAL
                             0
OF14 0000
        3764 WVAL
                                    ! VECTOR OF !
0F16 0000
             3765
                                    ! VECTOR 10 FOR SIO CHIP!
0F18 0000
         3766
                       WVAL
                             20000
OF1A 0000
             3767
                       HUAL
                             20000
                                     I VECTOR 11 I
OF1C 0000
        3768
                      NVAL 20000
                                     ! VECTOR 12 !
                                   ! VECTOR 13 !
! VECTOR 14 FOR CH.B KB !
! VECTOR 15 !
OF1E 0000
        3769 WVAL
                             0
                             0
0F20 0000
         3770
                       WVAL
0F22 0000
                       HVAL
                             0
             3771
                                    ! VECTOR 16 CHB SP.REC.COND.!
! VECTOR 17 !
           3772
                       WUAL
0F24 0000
                             0
0F26 0000
         3773
                       HVAL
                             0
                                   ! VECTOR 18 !
         3774
                      WVAL
                             0
0F28 0000
                                    ! VECTOR 19 !
         3775 WVAL
                             0
0F2A 0000
                            0 ! VECTOR 1A !
0 ! VCTORE 1B !
0 ! VECTOR 1C CH.A MCZ INPUT !
0F2C 0000
             3776 WVAL
                  WVAL
0F2E 0000
             3777
0F30 0000
             3778
                      WUAL
                            0
                       HVAL 0
HVAL 0
             3779 WVAL
                                    ! VECTOR 1D !
0F32 0000
                                    ! VECTOR 1E CHA SP.REC.COND !
! VECTOR 1F !
0F34 0000
             3780
0F36 0000
             3781
             3782
             3783
                  ! CTC VALUE
                                 BAUDE RATE SETTING !
             3784
                                  ! 110 (0) !
                              175
OF38 AF
             3785 SWTBL: BVAL
                              143 ! 134.5 (1) !
             3786 BVAL
0F39 8F
                              128 ! 150 (2) !
              3787
                        BVAL
0F3A 80
                              96 ! 200 (3) !
0F3B 60
             3788
                      BVAL
                                    ! 300 (4) !
! 600 (5) !
! 1200 (6) !
                  BVAL
OF3C 40
              3789
                              64
              3790
                   BVAL
                              32
0F3D 20
              3791
                        BVAL
                              16
OF3E 10
                              8 ! 2400 (7) !
                     BVAL
0F3F 08
              3792
                              4 ! 4800 (8) !
              3793
                        BVAL
0F40 04
          3794 BVAL
                              2 ! 9600 (9) !
0F41 02
```

```
0F42 01
                  3795
                                 BVAL
                                                   ! 19200 (A) !
0F43 01
                  3796
                                 BVAL
                                         1
                                                   ! 19200 (B) !
                                 BVAL
                                                   ! 19200 (C) !
0F44 01
                  3797
                                         1
0F45 01
                  3798
                                 BUAL
                                         1
                                                   ! 19200 (D) !
0F46 01
                  3799
                                 BUAL
                                         1
                                                   ! 19200 (E) !
                                 BUAL
                                                   ! 19200 (F) !
0F47 01
                  3800
                                         1
                  3801
                  3802
                  3803 $REL %0FA0
                  3804
                                JP
                                       TYIN
OFA0 5E08 075E'
                  3805
                                JP
                                       CONVERT
0FA4 5E08
          06FA'
                   3806
                                JP
                                       NXTARG
0FA8 5E08
           07481
                   3807
                                JP
OFAC SEOR
           08DA
                   3808
                                       1002
OFBO 5E08
           07221
                  3809
                                JP
                                       GETADN
                                JP
                                       PUT2
OFB4 5E08
           09321
                   3810
           0936
                   3811
                                JP
                                       PUT1
OFB8 5E08
OFBC 5E08
                                JP
                                       PUTPTR
           09761
                   3812
                                JP
OFCO 5E08
           08B47
                   3813
                                       PUTMSG
0FC4 5E08
           06561
                   3814
                                JP
                                       GETNUM
                                JP
                                       TYNR
OFCB SEOB
           07761
                   3815
                                JP
                                       PUTNCR
OFCC SEOR
           0982
                   3816
                                JP
                                       EROR
OFDO 5E08
           014E
                   3817
                                JP
                                       CRLF
0FD4 5E08
           07A0'
                   3818
                                JP
                                       MCZSND
OFD8 5E08
           078E'
                   3819
                                JP
                                       TTY
OFDC 5E08
           068A
                   3820
                                JP
                                       TTYRD
OFEO SEOB
           06F61
                   3821
0FE4 5E08
           0752
                   3822
                                JP
                                       SKIPSP
                                JP
                                       GCHAR
0FE8 5E08
           07AC'
                   3823
                                JP
                                       GET
OFEC SEO8
           07C01
                   3824
                   3825
                   3826
                   3827 END INTER
0FF0
                   3828
                   3829 END ZDISPLAY
```

Z8001/Z8002 CPU Product Specification

Z8001/Z8002 CPU

Z8001 CPU Z8002 CPU



Product Specification

Preliminary

Features

- ☐ Regular, easy-to-use architecture
- ☐ Instruction set more powerful than many minicomputers
- □ Directly addresses eight megabytes
- ☐ Eight user-selectable addressing modes
- □ Seven data types that range from bits to 32-bit long words and word strings
- □ System and normal operating modes
- ☐ Separate code, data and stack spaces
- □ Sophisticated interrupt structure

- ☐ Resource-sharing capabilities for multiprocessing systems
- ☐ Multi-programming support
- □ Compiler support
- ☐ Memory management and protection provided by Z8010 Memory Management Unit
- □ 32-bit operations, including signed multiply and divide
- □ Z-Bus compatible
- □ 4 MHz clock rate

General Description

The Z8000 is an advanced high-end 16-bit microprocessor that spans a wide variety of applications ranging from simple stand-alone computers to complex parallel-processing systems. Essentially, a monolithic minicomputer central processing unit, the Z8000 CPU is characterized by an instruction set more powerful than many minicomputers; resources abundant in registers, data types, addressing modes and addressing range; and a regular architecture that enhances throughput by avoiding critical bottlenecks such as implied or dedicated registers.

CPU resources include sixteen 16-bit general-purpose registers, seven data types that range from bits to 32-bit long words and word strings, and eight user-selectable addressing modes. The 110 distinct instruction types can be combined with the various data types and addressing modes to form a powerful set of 414 instructions. Moreover, the instruction set exhibits a high degree of regularity: most instructions can use any of the five main addressing modes and can operate on byte, word and long-word data types.

The CPU can operate in either the system or normal modes. The distinction between these two modes permits privileged operations, thereby improving operating system organization and implementation. Multiprogramming is supported by the "atomic" Test and Set instruction; multiprocessing by a combination

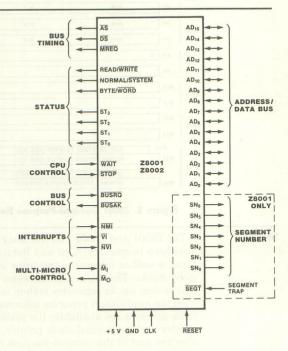


Figure 1. Z8000 CPU Pin Functions

of instruction and hardware features; and compilers by multiple stacks, special instructions and addressing modes.

General Description (Continued)

The Z8000 CPU is offered in two versions: the Z8001 48-pin segmented CPU and the Z8002 40-pin non-segmented CPU. The main difference between the two is in addressing range. The Z8001 can directly address 8 megabytes of memory; the Z8002 directly addresses 64 kilobytes. The two operating modes—system and normal—and the distinction between code, data and stack spaces within each mode allows memory extension up to 48 megabytes for the Z8001 and 384 kilobytes for the Z8002.

To meet the requirements of complex, memory-intensive applications, a companion

memory-management device is offered for the Z8001. The Z8010 Memory Management Unit manages the large address space by providing features such as segment relocation and memory protection. The Z8001 can be used with or without the Z8010. If used by itself, the Z8001 still provides an 8 megabyte direct addressing range, extendable to 48 megabytes.

The Z8001, Z8002 and Z8010 are fabricated with high-density, high-performance scaled n-channel silicon-gate depletion-load technology, and are housed in dual in-line packages.

Register Organization

The Z8000 CPU is a register-oriented machine that offers sixteen 16-bit general-purpose registers and a set of special system registers. All general-purpose registers can be used as accumulators and all but one as index registers or memory pointers.

Register flexibility is created by grouping and overlapping multiple registers (Figures 2)

and 3). For byte operations, the first eight 16-bit registers (R0...R7) are treated as sixteen 8-bit registers (RL0, RH0, ..., RL7, RH7). The sixteen 16-bit registers are grouped in pairs (RR0...RR14) to form 32-bit long-word registers. Similarly, the register set is grouped in quadruples (RQ0...RQ12) to form 64-bit registers.

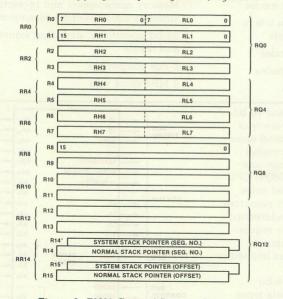


Figure 2. Z8001 General-Purpose Registers

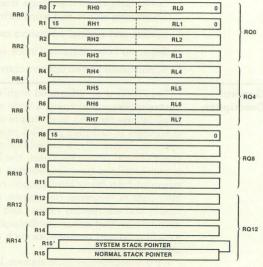


Figure 3. Z8002 General-Purpose Registers

Stacks

The Z8001 and Z8002 can use stacks located anywhere in memory. Call and Return instructions as well as interrupts and traps use implied stacks. The distinction between normal and system stacks separates system information from the application program information. Two stack pointers are available: the system stack pointer and the normal stack pointer. Because they are part of the general-purpose register

group, the user can manipulate the stack pointers with any instruction available for register operations.

In the Z8001, register pair RR14 is the implied stack pointer. Register R14 contains the 7-bit segment number and R15 contains the 16-bit offset. In the Z8002, register R15 is the implied 16-bit stack pointer.

Refresh

The Z8000 CPU contains a counter that can be used to automatically refresh dynamic memory. The refresh counter register consists of a 9-bit row counter, a 6-bit rate counter and an enable bit (Figure 4). The 9-bit row counter can address up to 256 rows and is incremented by two each time the rate counter reaches end-of-count. The rate counter determines the time between successive refreshes. It consists of a programmable 6-bit modulo-n prescaler

(n = 1 to 64), driven at one-fourth the CPU clock rate. The refresh period can be programmed from 1 to 64 μ s with a 4 MHz clock. Refresh can be disabled by programming the refresh enable/disable bit.

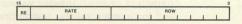


Figure 4. Refresh Counter

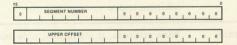
Program Status Information

This group of status registers contains the program counter, flags and control bits. When an interrupt or trap occurs, the entire group is saved and a new program status group is loaded.

Figure 5 illustrates how the program status groups of the Z8001 and Z8002 differ. In the non-segmented Z8002, the program status group consists of two words: the program counter (PC), and the flag and control word (FCW). In the segmented Z8001, the program



20001 Frogram Status Register



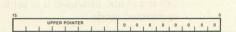
Z8001 Program Status Area Pointer

status group consists of four words: a two-word program counter, the flag and control word and an unused word reserved for future use. Seven bits of the first PC word designate one of the 128 memory segments. The second word supplies the 16-bit offset that designates a memory location within the segment.

With the exception of the segment enable bit in the Z8001 program status group, the flags and control bits are the same for both CPUs.



Z8002 Program Status Registers



Z8002 Program Status Area Pointer

Figure 5. Z8000 CPU Special Registers

Interrupt and Trap Structure

The Z8000 provides a very flexible and powerful interrupt and trap structure. Interrupts are external asynchronous events requiring CPU attention, and are generally triggered by peripherals needing service. Traps are synchronous events resulting from the execution of certain instructions. Both are processed in a similar manner by the CPU.

The CPU supports three types of interrupts (non-maskable, vectored and non-vectored) and four traps (system call, unimplemented instruction, privileged instructions and segmentation trap). The vectored and non-vectored interrupts are maskable. Of the four traps, the only external one is the segmentation trap, which is generated by the Z8010.

The remaining traps occur when instructions limited to the system mode are used in the normal mode, or as a result of the System Call instruction, or for an unimplemented instruction. The descending order of priority for traps

and interrupts is: internal traps, non-maskable interrupt, segmentation trap, vectored interrupt and non-vectored interrupt.

When an interrupt or trap occurs, the current program status is automatically pushed on the system stack. The program status consists of the processor status (PC and FCW) plus a 16-bit identifier. The identifier contains the reason or source, of the trap or interrupt. For internal traps, the identifier is the first word of the trapped instruction. For external traps or interrupts, the identifier is the vector on the data bus read by the CPU during the interrupt-acknowledge or trap-acknowledge cycle.

After saving the current program status, the new program status is automatically loaded from the program status area in system memory. This area is designated by the program status area pointer (PSAP).

Data Types

Z8000 instructions can operate on bits, BCD digits (4 bits), bytes (8 bits), words (16 bits), long words (32 bits), byte strings and word strings (up to 64 kilobytes long). Bits can be set, reset and tested; digits are used in BCD arithmetic operations; bytes are used for characters or small integer values; words are used for integer values, instructions and non-segmented addresses; long words are used for

long integer values and segmented audresses. All data elements except strings can reside either in registers or memory. Strings are stored in memory only.

The basic data element is the byte. The number of bytes used when manipulating a data element is either implied by the operation or for strings and multiple register operations – explicitly specified in the instruction.

Segmentation and Memory Management

High-level languages, sophisticated operating systems, large programs and data bases, and decreasing memory prices are all accelerating the trend toward larger memory requirements in microcomputer systems. The Z8001 meets this requirement with an eight

megabyte addressing space. This large address space is directly accessed by the CPU using a segmented addressing scheme and can be managed by the Z8010 Memory Management Unit.

Segmented Addressing

A segmented addressing space-compared with linear addressing-is closer to the way a programmer uses memory because each procedure and data space resides in its own segment. The 8 megabytes of Z8001 addressing space is divided into 128 relocatable segments up to 64 kilobytes each. A 23-bit segmented address uses a 7-bit segment address to point to the segment, and a 16-bit offset to address any location relative to the beginning of the seament. The two parts of the segmented address may be manipulated separately. The segmented Z8001 can run any code written for the non-segmented Z8002 in any one of its 128 segments, provided it is set to the nonsegmented mode.

In hardware, segmented addresses are contained in a register pair or long-word memory location. The segment number and offset can be manipulated separately or together by all the available word and long-word operations.

When contained in an instruction, a segmented address has two different representations: long offset and short offset. The long offset occupies two words, whereas the short offset requires only one and combines in one word the 7-bit segment number with an 8-bit offset (range 0-256). The short offset mode allows very dense encoding of addresses and minimizes the need for long addresses required by direct accessing of this large address space.

Memory Management

The addresses manipulated by the programmer, used by instructions and output by the Z8001 are called *logical* addresses. The Memory Management Unit takes the logical addresses and transforms them into the *physical* addresses required for accessing the memory (Figure 6). This address transformation process is called relocation. Segment relocation makes user software addresses independent of the physical memory so the user is freed from specifying where information is actually located in the physical memory.

The relocation process is transparent to user software. A translation table in the Memory Management Unit associates the 7-bit segment number with the base address of the physical memory segment. The 16-bit offset is added to the physical base address to obtain the actual physical address. The system may dynamically reload translation tables as tasks are created, suspended or changed.

In addition to supporting dynamic segment relocation, the Memory Management Unit also provides segment protection and other segment management features. The protection features prevent illegal uses of segments, such as writing into a write-protected zone.

Each Memory Management Unit stores 64 segment entries that consist of the segment

base address, it attributes, size and status. Segments are variable in size from 256 bytes to 64 kilobytes in increments of 256 bytes. Pairs of Management Units support the 128 segment numbers available for each of the six CPU address spaces. Within an address space, several Management Units can be used to create multiple translation tables.

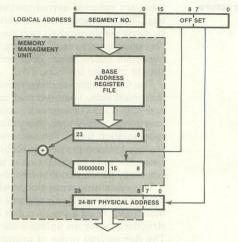


Figure 6. Logical-to-Physical Address Transformation

Addressing Modes

The information included in Z8000 instructions consists of the function to be performed, the type and size of data elements to be manipulated and the location of the data elements. Locations are designated by register addresses, memory addresses or I/O addresses. The addressing mode of a given instruction defines the address space it references and the method used to compute the address itself. Addressing modes are explicitly specified or implied by the instruction.

Figure 4 illustrates the eight addressing modes: Register (R), Immediate (IM), Indirect Register (IR), Direct Address (DA), Indexed (X), Relative Address (RA), Base Address (BA) and Base Indexed (BX). In general, an addressing mode explicitly specifies either register address space or memory address space. Program memory address space and I/O address space are usually implied by the instruction.

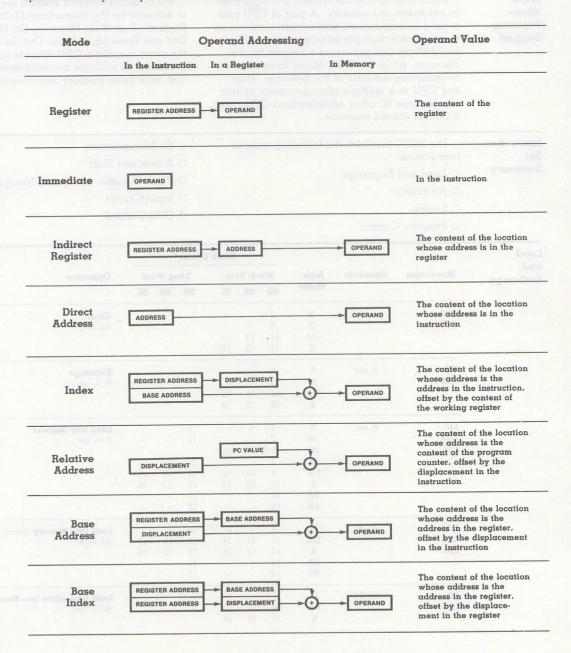


Figure 7. Addressing Modes

Input/ Output

A set of I/O instructions performs 8-bit or 16-bit transfers betwen the CPU and I/O devices. I/O devices are addressed with a 16-bit I/O port address. The I/O port address is similar to a memory address; however, I/O address space is not part of the memory address space. I/O port and memory addresses coexist on the same bus lines and they are distinguished by the status outputs.

Two types of I/O instructions are available: standard and special. Each has its own address space. Standard I/O instructions include a comprehensive set of In, Out and Block I/O instructions for both bytes and words. Special I/O instructions are used for loading and unloading the Memory Management Unit. The status information distinguishes between standard and special I/O references.

Multi-Micro-Processor Support

Multi-microprocessor systems are supported in hardware and software. A pair of CPU pins is used in conjunction with certain instructions to coordinate multiple microprocessors. The Multi-Micro Out pin issues a request for the resource, while the Multi-Micro In pin is used to recognize the state of the resource. Thus, any CPU in a multiple microprocessor system can exclude all other asynchronous CPUs from a critical shared resource.

Multi-microprocessor systems are supported in software by the instructions Multi-Micro Request, Test Multi-Micro In, Set Multi-Micro Out and Reset Multi-Micro Out. In addition, the eight megabyte CPU address space is beneficial in multiple microprocessor systems that have large memory requirements.

Instruction Set Summary

The Z8000 provides the following types of instructions:

- □ Load and Exchange
- □ Arithmetic
- □ Logical
- □ Program Control

- ☐ Bit Manipulation
- □ Rotate and Shift
- □ Block Transfer and String Manipulation
- □ Input/Output
- □ CPU Control

| Load | | | Āddr. | Clock Cycles* | | | | | | | |
|-----------------|------------------|----------|-------|---------------|---------|--------|------------|-------|--------|----------------------------|--|
| and Exchange | Mnemonics | Operands | | Word, Byte | | yte | te Lor | | ord | Operation | |
| | | | Modes | NS | SS | SL | NS | SS | SL | | |
| | CLR | dst | R | 7 | _ | _ | | H | | Clear | |
| | CLRB | | IR | 8 | _ | _ | | | | dst ← 0 | |
| | | | DA | 11. | 12 | 14 | | | | ast | |
| | | | X | 11 | 12 | 15 | | | | | |
| | EX | R, src | R | 6 | _ | - | | | 184 | Exchange | |
| | EXB | | IR | 12 | THE DAY | Sale - | | | | R ↔ src | |
| | | | DA | 15 | 16 | 18 | | | | and a second | |
| | regardent and | | X | 16 | 16 | 19 | | | | | |
| | LD | R, src | R | 3 | - | - | 5 | _ | _ | Load into Register | |
| | LDB | | IM | 7 | - | - | 11 | _ | - | R ← src | |
| | LDL | | IM | 5 (1 | oyte o | nly) | | | | | |
| | | | IR | 7 | _ | _ | 11 | - | - | | |
| | | | DA | 9 | 10 | 12 | 12 | 13 | 15 | | |
| | | | X | 10 | 10 | 13 | 13 | 13 | 16 | | |
| | | | BA | 14 | - | - | 17 | - | - | | |
| | ra tractaco oul? | | BX | 14 | - | - | 17 | - | - | | |
| | LD | dst, R | IR | 8 | - | | 11 | - | _ | Load into Memory (Store) | |
| | LDB | | DA | 11 | 12 | 14 | 14 | 15 | 17 | dst ← R | |
| | LDL | | X | 12 | 12 | 15 | 15 | 15 | 18 | | |
| | | | BA | 14 | - | - | 17 | - | - | | |
| | o Include off | | BX | 14 | - | - | 17 | - | - | | |
| | LD | dst, IM | IR | 11 | SANGE S | EAS - | A STATE OF | HA.RE | ALD BA | Load Immediate into Memory | |
| | LDB | | DA | 14 | 15 | 17 | | | | dst ← IM | |
| | | | Y | 15 | 15 | 10 | | | | | |

^{*} NS = Non-Segmented SS = Segmented Short Offset SL = Segmented Long Offset

| Load and | Mnemonics | Operands | | | | Clock | Cycles | WE HE | | | |
|------------------------|---|-------------|---------|------------|----------|----------|----------|----------|----------|---|--|
| Exchange Continued) | | | Addr. | Word, Byte | | | Lo | ng W | ord | Operation | |
| Commuca | | | Modes | NS | SS | SL | NS | SS | SL | | |
| | LDA | R, src | DA | 12 | 13 | 15 | H | | Siz. H | Load Address | |
| | | | X | 13 | 13 | 16 | | | | R ← source address | |
| | | | BA | 15 | - | 80_ | | | | | |
| | | | BX | 15 | 12 | 10- | | | | | |
| | LDAR | R, src | RA | 15 | - | 11 | 1 | | lab | Load Address Relative R ← source address | |
| | LDK | R, src | IM | 5 | - | - | | | | Load Constant | |
| | and transport | | | | | | | | | $R \leftarrow n (n = 0 15)$ | |
| | LDM | R, src, n | IR | 11 | - | - 1 | | | | Load Multiple | |
| | | | DA | 14 | 15 | 17 } | +3n | | | R ← src (n consecutive words) | |
| | | | X | 15 | 15 | 18) | | | | (n = 1 16) | |
| | LDM | dst, R, n | IR | 11 | - | 02 \ | 9 | | Sta R | Load Multiple (Store Multiple) | |
| | MAP AND | GOL, 11, 11 | DA | 14 | 15 | 17 | +3n | | | dst ← R (n consecutive words) | |
| | | | X | 15 | 15 | 18 | Al | | | $(n = 1 \dots 16)$ | |
| | LDR | R, src | RA | 14 | 85 | 50 | 17 | - | - | Load Relative | |
| | LDRB LDRL | | | | | | | | | $R \leftarrow src$ (range -32768 + 32767) | |
| | LURL | | | | | | - m | ye'n. | | (range -52700 + 52707) | |
| | LDR | dst, R | RA | 14 | BL a | EL. | 17 | - | 9-1-10 | Load Relative (Store Relative) | |
| | LDRB | | | | | | | | | dst ← R | |
| | LDRL | | | | | | | | - F | (range -32768 + 32767) | |
| | POP | dst, IR | R | 8 | - | - | 12 | - | - | Pop | |
| | POPL | | IR | 12 | - | - | 19 | - | - | dst ← IR | |
| | | | DA | 15 | 16 | 18 | 22 | 23 | 25 | Autoincrement contents of R | |
| | | | X | 16 | 16 | 19 | 23 | 23 | 26 | | |
| | PUSH | IR, src | R | 9 | OL | e_ | 12 | - | | Push | |
| | PUSHL | 61 31 | IM | 12 | OL | 05 | 1 | _ | _ | Autodecrement contents of R | |
| | | | IR | 13 | - | - | 20 | - | - | IR ← src | |
| | | | DA | 13 | 14 | 16 | 20 | 21 | 23 | | |
| | | | X | 14 | 14 | 17 | 21 | 21 | 24 | Logical AND | |
| Arithmetic | ADC ADCB | R, src | R | 5 | 01 | 6 - | AD AQ | | | Add with Carry R ← R + src + carry | |
| | | | | | OL | DL | - A | | | | |
| | ADD | R, src | R | 4 | - | - | 8 | - | dst | Add P P ara | |
| | ADDB | | IM | 7 | - | SI | 14 | - | 140 | $R \leftarrow R + src$ | |
| | ADDL | | IR | 7 | 10 | 10 | 14 | 16 | 10 | | |
| | | | DA X | 9 | 10 | 12 13 | 15 16 | 16 16 | 18 19 | | |
| | - 80 | | | | | h . | | | 279 11 | a Was | |
| | CP | R, src | R | 4 | - | 7- | 8 | - | | Compare with Register | |
| | CPB | | IM | 7 | - | 1 | 14 | - | - | R - src | |
| | CPL | | IR | 7 | 010 | 12 | 14 | 16 | 19 | | |
| | | | DA X | 9 | 10 | 12 13 | 15 16 | 16 16 | 18 19 | | |
| | Test | | | | | - | T II | | teb | 1277 | |
| | CP | dst, IM | IR | 11 | - | - | | | | Compare with Immediate | |
| | CPB | | DA X | 14 | 15 15 | 17 18 | | | | dst - IM | |
| | | 17 20 | | | 10 | | X | | | | |
| | DAB | dst | R | 5 | - | 27 | Я | | ish ou | Decimal Adjust | |
| | DEC | dst, n | R | 4 | - | - | | | | Decrement by n | |
| | | | IR | 11 | - | - | | | | dst ← dst - n | |
| | DECB | | 111 | | | | | | | | |
| | DECB | | DA X | 13 14 | 14 14 | 16 17 | | | | (n = 1 16) | |

| Arithmetic | | | Clock Cycles | | | | | | | |
|-------------|-----------|----------|--|----------|----------|----------|------------|----------|------------|--|
| (Continued) | Mnemonics | Operands | Addr. | Wo | ord, B | yte | Lon | g Wo | ord | Operation |
| | | | Modes | NS | SS | SL | NS | SS | SL | |
| | DIV | R, src | R | 95 | - | S- | 723 | _ | ine A | Divide (signed) |
| | DIVL | | IM | 95 | - | EL | 723 | - | - | Word: $R_{n+1} \leftarrow R_{n,n+1} \div src$ |
| | | | IR | 95 | - 07 | - | 723 | - | - | $R_n \leftarrow remainder$ |
| | | | DA X | 96 97 | 97 97 | 99 | 724 725 | 725 | 727 728 | Long Word: $R_{n+2,n+3} \leftarrow R_{n,n+3} \div s$ $R_{n,n+1} \leftarrow remainder$ |
| | EXTS | dst | R | 11 | - | _ | 11 | - | 301.71 | Extend Sign |
| | EXTSB | | | | | | | | | Extend sign of low order half of dst |
| | EXTSL | | | | | 8 | MI | | Regre | through high order half of dst |
| | INC | dst, n | R | 4 | _ | - | | | | Increment by n |
| | INCB | | IR | 11 | - | - | | | d Sat H | $dst \leftarrow dst + n$ |
| | | | DA X | 13 14 | 14 | 16 17 | | | | (n = 1 16) |
| | | | The state of the s | | 11 | -17 | | | | |
| | MULT | R, src | R | -70 | - | 15 | 282* | | dat litt | Multiply (signed) |
| | MULTL | | IM | 70 70 | - | 44 | 282* | | - | Word: $R_{n,n+1} \leftarrow R_{n+1} \cdot \text{src}$ |
| | | | IR DA | 71 | 72 | 74 | 282* | | 286* | Long Word: $R_{nn+3} \leftarrow R_{n+2}$, $n+3$. *Plus seven cycles for each 1 in the |
| | | | X | 72 | 72 | 75 | | | 287* | multiplicand |
| | NEG | dst | R | 7 | | | | | | Negate |
| | NEGB | ust | IR | 12 | _ | _ | | | | dst ← 0 - dst |
| | | | DA | 15 | 16 | 18 | | | | |
| | H — let | | X | 16 | 16 | 19 | | | | arga . |
| | SBC | R, src | R | 5 | - | - | | | | Subtract with Carry |
| | SBCB | | LI . | | | 8. | 8 | | H lak | R ← R - src - carry |
| | SUB | R, src | R | 4 | - | - | 8 | - | - | Subtract |
| | SUBB | | IM | 7 | 95 | 37 | 14 | - | - | R ← R - src |
| | SUBL | | IR | 7 | - | - | 14 | - | - | |
| | | | DA X | 9 | 10 | 12 | 15 16 | 16 16 | 18 19 | |
| | ine — Al | | 1205 | | | ti | Al | | | THEOR |
| Logical | AND | R, src | R | 4 | 1- | | ATI | | | AND |
| | ANDB | | IM | 7 | - | - | | | | R ← R AND src |
| | | | IR | 7 | - | - | | | | |
| | | | DA X | 9 | 10 | 12 13 | | | | |
| | COM | dst | R | 7 | _ | + | A | | marif | Complement |
| | COMB | usi | IR | 12 | _ | | | | | dst ← NOT dst |
| | | | DA | 15 | 16 | 18 | | | | |
| | | el al | X | 16 | 16 | 19 | X | | | |
| | OR | R, src | R | 4 | - | - | | | | OR |
| | ORB | | IM | 7 | - | 7- | | | | R ← R OR src |
| | | | IR | 7 | - | - | | | | |
| | | | DA X | 9 | 10 | 12 13 | | | | |
| | TEST | dat | | 7 | WI. | - 44 | 12 | | | Test |
| | TESTB | dst | R IR | 8 | | | 13 13 | _ | MILTED | dst OR 0 |
| | TESTL | | DA | 11 | 12 | 14 | 16 | 17 | 19 | 140 |
| | | | X | 12 | 12 | 15 | 17 | 17 | 20 | |
| | TCC | cc, dst | R | 5 | - | 2 | H | | lab i | Test Condition Code |
| | TCCB | | | | | p. | H | | if tab | Set LSB if cc is true |
| | XOR | R, src | R | 4 | - | - | | | | Exclusive OR |
| | XORB | | IM | 7 | 41- | 11- | | | | R ← R XOR src |
| | | | IR DA | 7 9 | 10 | 12 | | | | |
| | | | V | 10 | 10 | 12 | | | | |

| Program | | | | Clock | | Clo | ck (| Cycles | | | |
|-----------|------------------------------|----------|-----------|------------|----------|-------|------|--------|-------|--------------------|---|
| Control | Mnemonics | Operands | Addr. | Wo | rd, B | yte | | Long | g Wo | rd | Operation |
| | | | Modes | es NS SS S | | SL | | NS | SS | SL | |
| | CALL | dst | IR | 10 | - | 15 | 5 | , date | | na A | Call Subroutine |
| | | | DA X | 12 | 18 | 20 | | | | | Autodecrement SP @ SP ← PC |
| | | | Λ | 10 | 10 | 2.1 | | | | | PC ← dst |
| | CALR | dst | RA | 10 | 211 | 15 | ; | | 301 | | Call Relative |
| | | usi | IIA | | | | | | | | Autodecrement SP |
| | | | | | | | | | | | @ SP ← PC PC ← PC + dst (range -4094 to +4096 |
| | | | | | | | | | | | Te Te Tast (lange 1001 to Tibe |
| | DJNZ DBJNZ | R, dst | RA | | - 113 | | | | | | Decrement and Jump if Non-Zero R ← R − 1 |
| | DBJNZ | | | | | | | | | | If $R \neq 0$: PC \leftarrow PC + dst (range -254 to 0 |
| | IRET* | | | 13 | | 16 | | A | | | Interrupt Return |
| | | | | | | | , | | | | PS ← @ SP |
| | | | + 80 | (a | | ar) | | H | | A _s teb | Autoincrement SP |
| | JP | cc, dst | IR | 10 | - | 15 | 5 | (ta | aken) | | Jump Conditional |
| | | | IR DA | 7 | - 8 | 10 | | (not | take | en) | If cc is true: PC ← dst |
| | | | X | 8 | 8 | 11 | | | | | |
| | N la stration of | 1-1 | RA | 6 | | | | | | | Jump Conditional Relative |
| | JR | cc, dst | nA - E | | | 81) | | | | | If cc is true: PC ← PC + dst |
| | | | | | | | | | | | (range -256 to +254) |
| | RET | CC | - | 10 | - | 13 | 3 | (ta | aken) | | Return Conditional |
| | | | | 7 | - | 7 | 7 | (not | take | en) | If cc is true: PC ← @ SP Autoincrement SP |
| | | | | | | | | | | | Automorement of |
| | SC | src | IM | 33 | - | 39 | 9 | | | | System Call Autodecrement SP |
| | | | | | | | | | | | @ SP ← old PS |
| | | | | | | | | | | | Push instruction |
| | | in B | + E1) | (n | E E | e i i | | А | | in July | PS ← System Call PS |
| Bit | BIT | dst, b | R | 4 | - | _ | | | | | Test Bit Static |
| Manipula- | BITB | | IR DA | 8 | 11 | 1 | 3 | | | | Z flag ← NOT dst bit specified by b |
| tion | | | X | 11 | 11 | 1 | | | | | |
| | BIT | -1-4 D | R | 10 | | | | | | | Test Bit Dynamic |
| | BITB | dst, R | n | 10 | | | | | | | Z flag — NOT dst bit specified by |
| | | | | | | FILE | | 91 | | | contents of R |
| | RES | dst, b | R | 4 | - | | | | | | Reset Bit Static |
| | RESB | | IR | 11 | 14 | 1 | | | | | Reset dst bit specified by b |
| | | | DA X | 13 14 | 14 14 | 1 | | | | | |
| | Compare Comp | dat D | R | 10 | | 01 | | HI | 79.5 | old man | Reset Bit Dynamic |
| | RES RESB | dst, R | п | 10 | | | | | | | Reset dst bit specified by contents R |
| | SET | dat h | R | 4 | | | | | | | Set Bit Static |
| | SETB | dst, b | IR | 11 | - | - | | | | | Set dst bit specified by b |
| | | | DA X | 13 14 | 14 14 | 1 | | | | | |
| | or <u>a tresmenoruciti A</u> | | Λ | | 14 | 1 | 1 | | ne e | | |
| | SET | dst, R | R | 10 | | | | | | | Set Bit Dynamic |

dst

R

IR

DA

X

7

11

14 15 15 17

18

SETB

TSETB

Test and Set

S flag ← MSB of dst dst ← all ls

^{*}Privileged instruction, Executed in system mode only.

| and | Mnemonics C | perands | Addr. | Word | l. By | e | Lon | g W | ord | Operation | | |
|----------------|----------------------|--------------|--------|-------------|-------|-----|------|------|---------|--|--|--|
| Shift | | 19 88 | Modes | | 444 | SL | NS | SS | SL | le . surmanianis | | |
| | RLDB | R, src | R | 9 | - | - | M | | late | Rotate Digit Left | | |
| | RRDB | R, src | R | 9 | 7 | - | - 80 | | | Rotate Digit Right | | |
| | RL | dst, n | R | 6 for | n = 1 | | | | | Rotate Left | | |
| | RLB install lieb | | R | 7 for | n = 2 | 2 | All | | Tab. | by n bits $(n = 1, 2)$ | | |
| | RLC RLCB | dst, n | R R | 6 for 7 for | | | | | | Rotate Left through Carry by n bits $(n = 1, 2)$ | | |
| | RR | dst, n | R | 6 for | n = 1 | | 1 | | | Rotate Right | | |
| | RRB | dot, ii | R | 7 for | | | | | | by n bits $(n = 1, 2)$ | | |
| | RRC | dst, n | R | 6 for | | | | | | Rotate Right through Carry | | |
| | RRCB | | R | 7 for | n = 2 | | | | | by n bits (n = 1, 2) | | |
| | SDA SDAB | dst, R | R | (15 | + 3 n | 1) | (15 | + 3 | n) | Shift Dynamic Arithmetic Shift dst left or right | | |
| | SDAL | (C) Side | 00 | | | 915 | | | reb mi | by contents of R | | |
| | SDL SDLB | dst, R | R | (15 | + 3 n | 1) | (15 | + 3 | n) | Shift Dynamic Logical Shift dst left or right | | |
| | SDLL | | | | | | | | | by contents of R | | |
| | SLA | dst, n | R | (13 - | + 3 n | 1) | (13 | 3 +3 | n) | Shift Left Arithmetic | | |
| | SLAB SLAL | | | | | | | | | by n bits | | |
| | SLL | dst, n | R | (13 | + 3 r | 1) | (13 | + 3 | n) | Shift Left Logical | | |
| | SLLB SLLL | | | | | | | | | by n bits | | |
| | SRA | dst, n | R | (13 - | + 3 n | 1) | (13 | + 3 | n) | Shift Right Arithmetic | | |
| | SRAB SRAL | | | | | | | | | by n bits | | |
| | SRL | dat n | R | /12 | 1 2 5 | .) | (12 | + 3 | | Chife Dight Logical | | |
| | SRLB | dst, n | n | (13 | + 3 n | 1) | (13 | + 3 | n) | Shift Right Logical by n bits | | |
| ditailorge (s | SRLL | | | | | 9 | 41 | | | BTIE -placingM | | |
| ock ansfer | | src, Ry, cc | IR | 20 | - | - | | | | Compare and Decrement | | |
| d String | CPDB | | | | | | | | | R _X - src Autodecrement src address | | |
| anipula- on | lab (13) = pril A | | | | | | | | | Ry ← Ry - l | | |
| | CPDR R _X | src, Ry, cc | IR | (11 - | + 9 n |) | | | | Compare, Decrement and Repeat R _X - src | | |
| | Michael Helder Todal | | | | | | | | | Autodecrement src address | | |
| | | | | | | | | | | $R_Y \leftarrow R_Y - 1$ Repeat until cc is true or $R_Y = 0$ | | |
| | CPI R _X | src, Ry, cc | IR | 20 | | - | | | 1.14 | Compare and Increment | | |
| | CPIB | | | | | | | | | R _X – src Autoincrement src address | | |
| | Set Bit State | | | | | | T. | | al self | $R_Y \leftarrow R_Y - 1$ | | |
| | | src, Ry, cc | IR | (11 - | + 9 n |) | Aci | | | Compare, Increment and Repeat | | |
| | CPIRB | | | | | | | | | R _X - src Autoincrement src address | | |
| | | | | | | | | | | $R_Y \leftarrow R_Y - 1$ Repeat until cc is true or $R_Y = 0$ | | |
| | CPSD ds | , src, R, cc | IR | 25 | | _ | | | | Compare String and Decrement | | |
| | CPSDB | | | | | | | | | dst - src Autodecrement dst and src address | | |
| | | | | | | | | | | | | |

Autodecrement dst and src addresses R - R - 1

| Block Transfer | | | | | Clock | k Cycles | | | |
|-------------------------------------|--|-----------------|----------------|------|-------------------|----------|------|------------|--|
| and String Manipulation (Continued) | Mnemonics | Operands | Addr. Modes | Wo | rd, Byte SS SL | Lor | ng W | ord SL | Operation paid Date of the particular of the par |
| | CPSDR CPSDRB | dst, src, R, cc | IR | (11 | + 14 n) | Я | II | ero2, R | Compare String, Decr. and Repeat dst - src Autodecrement dst and src addresses R - R - 1 Repeat until cc is true or R = 0 |
| | CPSI CPSIB | | IR | 25 | | S. A | | A Son | Compare String and Increment dst – src Autoincrement dst and src addresses R + R – 1 |
| | CPSIR CPSIRB | | IR | (11) | + 14 n) | Я | | Halisma | Compare String, Incr. and Repeat dst - src Autoincrement dst and src addresses R - R - 1 Repeat until cc is true or R = 0 |
| | LDD LDDB | dst, src, R | IR | 20 | 6 | | | эн Я.эн | Load and Decrement dst — src Autodecrement dst and src addresses R — R - 1 |
| | LDDR LDDRB | dst, src, R | IR | | + 9 n) | | | | Load, Decrement and Repeat dst — src Autodecrement dst and src addresses R — R - 1 |
| | LDI | dst, src, R | IR | 20 | a 01 + (1) | H H | | H ,ots | Repeat until R = 0 Load and Increment |
| | LDIB | | III | 20 | | | | | dst — src Autoincrement dst and src addresses R — R - 1 |
| | LDIR LDIRB | dst, src, R | IR | (11 | + 9 n) | | | | Load, Increment and Repeat dst — src Autoincrement dst and src addresses |
| | at, Increment or | gal reb | | | m01 + 1() | Я | | B,ou | $R \leftarrow R - 1$ Repeat until R = 0 |
| | TRDB | | IR | 25 | | | | | Translate and Decrement dst ← src (dst) Autodecrement dst address R ← R − 1 |
| | TRDRB | dst, src, R | IR | (11 | + 14 n) | 1 A | g , | | Translate, Decrement and Repeat |
| | tput and Decreased are specification and areased areas | | | | | | | | dst — src (dst) Autodecrement dst address R — R - 1 Repeat until R = 0 |
| | TRIB | dst, src, R | IR | 25 | a (01 + 7(1) | R | | R tota | Translate and Increment dst — src (dst) Autoincrement dst address R — R – 1 |
| | TRIRB | dst, src, R | IR | (11 | + 14 n) | | | | Translate, Increment and Repeat |

25

TRTDB

srcl, src2, R

dst ← src (dst)
Autoincrement dst address

Translate and Test, Decrement RH1 ← src 2 (src 1)

Autodecrement src 1 address R ← R - 1

 $R \leftarrow R - 1$ Repeat until R = 0

| Block Transfer | | | Clock C | Cycles | Operation Daniel San | | | |
|----------------------------|-----------------------------|----------|-------------|--|--|--|--|--|
| and String Manipulation | Mnemonics Operands | Addr. | Word, Byte | Long Word | | | | |
| (Continued) | JE | Modes | NS SS SL | NS SS SL | nensista (| | | |
| | TRTDRB srcl, src2, R | IR | (11 + 14 n) | | Translate and Test, Decr. and Repeat RH1 — src 2 (src 1) Autodecrement src 1 address R — R - 1 Repeat until R = 0 or RH1 = 0 | | | |
| | TRTIB src1, src2, R | IR | 25 | R1 | Translate and Test, Increment RH1 ← src 2 (src 1) Autoincrement src 1 address R ← R - 1 | | | |
| | TRTIRB src1, src1, src2, R | IR IR | (11 + 14 n) | AT SA | Translate and Test, Incr. and Repeat RH1 — src 2 (src 1) Autoincrement src 1 address R — R - 1 Repeat until R = 0 or RH1 = 0 | | | |
| input/ Output | IN* R, src INB* | IR DA | 10 12 | 21 21 21 21 21 21 21 21 21 21 21 21 21 2 | Input R → src | | | |
| | IND* dst, src, R INDB* | IR | 21 | en a | Input and Decrement dst — src Autodecrement dst address R — R – 1 | | | |
| | INDR* dst, src, R INDRB* | IR | (11 + 10 n) | nt i n | Input, Decrement and Repeat dst ← src Autodecrement dst address R ← R - 1 Repeat until R = 0 | | | |
| | INI* dst, src, R INIB* | IR | 21 | HI H | Input and Increment dst — src Autoincrement dst address R — R – 1 | | | |
| | INIR* dst, src, R INIRB* | IR | (11 + 10 n) | я R. | Input, Increment and Repeat dst — src Autoincrement dst address R — R - 1 Repeat until R = 0 | | | |
| | OUT* dst, R | IR DA | 10 12 | H R | Output dst ← R | | | |
| | OUTD* dst, src, R OUTDB* | IR | 21 | | Output and Decrement dst — src Autodecrement src address R — R – 1 | | | |
| | OTDR* dst, src, R OTDRB* | IR | (11 + 10 n) | HI H | Output, Decrement and Repeat dst — src Autodecrement src address R — R - 1 Repeat until R = 0 | | | |
| | OUTI* dst, src, R | IR | 21 | HI A | Output and Increment dst — src Autoincrement src address R — R - 1 | | | |
| | OTIR* dst, src, R OTIRB* | IR | (11 + 10 n) | AT A.St | Output, Increment and Repeat dst — src Autoincrement src address R — R - 1 | | | |

^{*}Privileged instructions. Executed in system mode only.

| nput/Output | | | | Clock | Cycles | | | |
|----------------|-------------------|--------------------|---------------------|--------------|--|--|--|--|
| Continued) | Mnemonics | Operands | Addr. | Word, Byte | Long Word | Operation | | |
| | | | Modes | NS SS SL | NS SS SL | (Dammed) | | |
| | SIN* SINB* | R, src | DA | 12 | H mil.H | Special Input R ← src | | |
| | SIND* SINDB* | dst, src, R | IR | 21 | | Special Input and Decrement dst — src Autodecrement dst address | | |
| | | | | 81 25 | MI pr | R ← R - 1 | | |
| | SINDR* SINDRB* | dst, src, R | IR | (11 + 10 n) | AG X | Special Input, Decrement and Repeated to the src | | |
| | | | | | | Autodecrement dst address R ← R - 1 Repeat until R = 0 | | |
| | SINI* SINIB* | dst, src, R | IR | 21 | R 16 | Special Input and Increment dst src | | |
| | l-Micro Set | | | | | Autoincrement dst address R — R - 1 | | |
| | SINIR* SINIRB* | dst, src, R | IR | (11 + 10 n) | | Special Input, Increment and Repeat dst — src | | |
| | | | | | | Autoincrement dst address $R \leftarrow R - 1$ Repeat until $R = 0$ | | |
| | SOUT* SOUTB* | dst, src | DA | 12 | pa Bumanah magambanah | Special Output dst - src | | |
| | SOUTD* | dst, src, R | IR | 21 | paintel | Special Output and Decrement | | |
| | 300100 | | | | Iwaya Jalee Jwaya true stoo | Autodecrement src address R ← R – 1 | | |
| | SOTDR* | dst, src, R | IR | (11 + 10 n) | oreg to grey o Carry | Special Output, Decr. and Repeat | | |
| | | | | | | Autodecrement src address R - R - 1 Repeat until R = 0 | | |
| | SOUTI* SOUTIB* | dst, src, R | IR | 21 | walitava a walitava a | Special Output and Increment dst - src | | |
| | | 0 = 1919 | E (N) | (beapte | reity is odd caser than or oqual l | Autoincrement src address R ← R - 1 | | |
| | SOTIR* | dst, src, R | R | (11 + 10 n) | | Special Output, Incr. and Repeat dst — src Autoincrement src address | | |
| | | | | | | R \leftarrow R - 1 Repeat until R = 0 | | |
| CPU Control | COMFLG | flags | all vacuud (54%) | v 7 gr = v - | rinebrayan sebso dar 1630 = OUTJU = | Complement Flag (Any combination of C, Z, S, P/V) | | |
| | DI* | int | ET ₃ -ST | 6 | Definition | Disable Interrupt (Any combination of NVI, VI) | | |
| | EI* | int | 1001- | 6 | nambs: | Enable Interrupt (Any combination of NVI, VI) | | |
| | HALT* | memori Justinos | 0011 | (8 + 3 n) | apbelwontos yen | HALT | | |
| | LDCTL* | CTLR, src | R | 7 | | Load into Control Register CTLR — src | | |
| | LDCTL* | dst, CTLR | R | 7 | | Load from Control Register dst ← CTLR | | |

567

| CPU | | | | 1 | - | Clock C | Cycles | | | | |
|-------------|----------------------------------|---------------------------|----------------|---------|--------|-----------|---------|-----------|---------------------|--|-------------------------------------|
| (Continued) | Mnemonics | Operands | Addr. Modes | W NS | ord, B | yte SL | Long W | ord SL | Operation | | |
| | LDCTLB | FLGR, src | R | 7 | - | - 41 | Ad | ane. | | into Flag Byte R R ← src | legister |
| | LDCTLB | dst, FLGR | R | 7 | - | 9 L | An | Alph | | from Flag Byte I - FLGR | Register |
| | LDPS* | src | IR | 12 | _ | 16 | | | Load | Program Status | |
| | | | DA | 16 | 20 | 22 | | | PS - | | |
| | | | X | 17 | 20 | 23 | | | | | |
| | MBIT* | Solu A. — A some fl | - | 7 | - | - | | | | Multi-Micro Bit if \overline{M}_{I} is Low; rese | et S if \overline{M}_{l} is High. |
| | MREQ* | dst | R | (1 | 2 + 7 | n) | | | Multi-Micro Request | | |
| | MRES* | - 165 | - | 5 | - | - | | | Multi | -Micro Reset | |
| | MSET * | - | - | 5 | - | - | | | Multi | -Micro Set | |
| | NOP | lorda _ | - | 7 | (H_D)* | ngin. | H | A Some | No O | peration | |
| | RESFLG | flag | - | 7 | - | - | | | Reset (Any | Flag combination of C | , Z, S, P/V) |
| | SETFLG | flag | - | 7 | - | | AC | 516 | Set F | lag combination of C | Z. S. P/V) |
| | *Privileged inst | ructions. Executed | in system n | node or | nly. | | | | | *aruoa | |
| Condition | Code | Meaning | | | | | Flag | Settings | | | CC Field |
| Codes | that there to be considered that | Always fa | lse | | | | _ | | | HUIUOK | 0000 |
| | | Always tr | ue | | | | | | | | 1000 |
| | Z | Zero | | | | | Z = | | | | 0110 |
| | NZ C | Not zero | | | | | Z = | | | | 1110 |
| | NC | Carry No Carry | | | | | C = C = | | | | 0111 |
| | PL | Plus | | | | | S = | | | | 1111 |
| | MI | Minus | | | | | S = | | | | 0101 |
| | NE | Not equal | | | | | Z = | | | | 1110 |
| | EQ | Equal | | | | | Z = Z = | | | | 0110 |
| | OV | Overflow | | | | | P/V | | | | 0100 |

| Condition | Code | Meaning | Flag Settings | SOUT | CC Field |
|-----------|------|--------------------------------|-----------------------------|------|----------|
| Codes | | Always false | | TUDE | 0000 |
| | | Always true | | | 1000 |
| | Z | Zero | Z = 1 | | 0110 |
| | NZ | Not zero | Z = 0 | | 1110 |
| | C | Carry | C = 1 | | 0111 |
| | NC | No Carry | C = 0 | | 1111 |
| | PL | Plus | S = 0 | | 1101 |
| | MI | Minus | S = 1 | | 0101 |
| | NE | Not equal | Z = 0 | | 1110 |
| | EQ | Equal | Z = 1 | | 0110 |
| | | Overflow | P/V = 1 | | 0100 |
| | NOV | No overflow | P/V = 0 | | 1100 |
| | PE | Parity is even | P/V = 1 | | 0100 |
| | PO | Parity is odd | P/V = 0 | | 1100 |
| | GE | Greater than or equal (signed) | (S XOR P/V) = 0 | | 1001 |
| | LT | Less than (signed) | (S XOR P/V) = 1 | | 0001 |
| | GT | Greater than (signed) | | | 1010 |
| | LE | Less than or equal (signed) | [Z OR (S XOR P/V)] = 1 | | 0010 |
| | UGE | Unsigned greater than or equal | C = 0 | | 1111 |
| | ULT | Unsigned less than | C = 1 | | 0111 |
| | UGT | Unsigned greater than | [(C = 0) AND (Z = 0)] = 1 | | 1011 |
| | ULE | Unsigned less than or equal | (C OR Z) = 1 | | 0011 |

Note that some condition codes have identical flag settings and binary fields in the instruction: Z = EQ, NZ = NE, C = ULT, NC = UGE, OV = PE, NOV = PO

| Status Line | ST ₃ -ST ₀ Definition | ST ₃ -ST ₀ | Definition |
|----------------|--|----------------------------------|-------------------------------|
| Codes | 0000 Internal operation | 1000 | Data memory request |
| | 0 0 0 1 Memory refresh | 1001 | Stack memory request |
| | 0 0 1 0 I/O reference | 1010 | Reserved |
| | 0011 Special I/O reference (e.g., to an MMU) | 1011 | Reserved |
| | 0 1 0 0 Segment trap acknowledge | 1100 | Program reference, nth word |
| | 0 1 0 1 Non-maskable interrupt acknowledge | 1101 | Instruction fetch, first word |
| | 0 1 1 0 Non-vectored interrupt acknowledge | 1110 | Reserved |
| | 0 1 1 1 Vectored interrupt acknowledge | 1111 | Reserved |

Pin Description **AD₀-AD₁₅.** Address/Data (inputs/outputs, active High, 3-state). These multiplexed address and data lines are used both for I/O and to address memory.

AS. Address Strobe (output, active Low, 3-state). The rising edge of AS indicates addresses are valid.

BUSAK. Bus Acknowledge (output, active Low). A Low on this line indicates the CPU has reliquished control of the bus.

BUSRQ. Bus Request (input, active Low). This line must be driven Low to request the bus from the CPU.

DS. Data Strobe (output, active Low, 3-state). This line times the data in and out of the CPU.

MREQ. Memory Request (output, active Low, 3-state). A Low on this line indicates that the address/data bus hold a memory address.

M_I, **M**_O. *Multi-Micro In, Multi-Micro Out* (input and output, active Low). These two lines form a resource-request daisy chain that allows one CPU in a multi-microprocessor system to access a shared resource.

NMI. Non-Maskable Interrupt (edge triggered, input, active Low). A high-to-low transition on NMI requests a non-maskable interrupt. The NMI interrupt has the highest priority of the three types of interrupts.

NVI. Non-Vectored Interrupt (input, active Low). A Low on this line requests a non-vectored interrupt.

CLK. System Clock (input). CLK is a 5V single-phase time-base input.

RESET. Reset (input, active Low). A Low on this line resets the CPU.

 $\mathbf{R}/\overline{\mathbf{W}}$. Read/Write (output, Low = Write, 3-state). R/ $\overline{\mathbf{W}}$ indicates that the CPU is reading from or writing to memory or I/O.

SN₀-SN₆. Segment Number (outputs, active High, 3-state). These lines provide the 7-bit segment number used to address one of 128 segments by the Z8010 Memory Management Unit. Output by the Z8001 only.

SEGT. Segment Trap (input, active Low). The Memory Management Unit interrupts the CPU with a Low on this line when the MMU detects a segmentation trap.

ST₀-ST₃. Status (outputs, active High, 3-state). These lines specify the CPU status (see table).

STOP. Stop (input, active Low). This input can be used to single-step instruction execution.

VI. Vectored Interrupt (input, active Low). A Low on this line requests a vectored interrupt.

WAIT. Wait (input, active Low). This line indicates to the CPU that the memory or I/O device is not ready for data transfer.

B/W. Byte/Word (output, Low = Word, 3-state). This signal defines the type of memory reference on the 16-bit address/data bus.

N/S. Normal/System Mode (output, Low = System Mode, 3-state). N/S indicates the CPU is in the normal or system mode.

Decouple. Output from on-chip negative substrate-bias generator. Presently not connected.

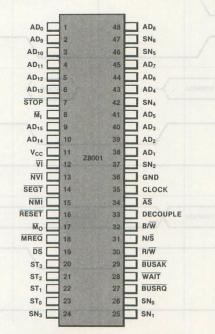


Figure 8. Z8001 Pin Assignments

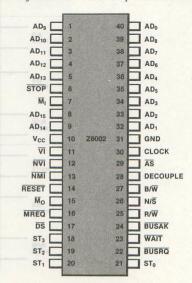


Figure 9. Z8002 Pin Assignments

Z8000 CPU Timing The Z8000 CPU executes instructions by stepping through sequences of basic machine cycles, such as memory read or write, I/O device read or write, interrupt acknowledge and internal execution. Each of these basic cycles require three to ten clock cycles to execute. Instructions that require more clock cycles to execute are broken up into several machine cycles. Thus no machine cycle is longer than ten clock cycles and fast response to a Bus Request is guaranteed.

The instruction opcode is fetched by a normal memory read operation. A memory refresh cycle can be inserted just after the completion of any first instruction fetch (IF₁)

cycle and can also be inserted while the following instructions are being executed: MULT, MULTL, DIV, DIVL, HALT, all Shift instructions, all Block Move instructions, and the Multi-Micro Request instruction (MREQ).

The following timing diagrams show the relative timing relationships of all CPU signals during each of the basic operations. When a machine cycle requires additional clock cycles for CPU internal operation, one to five clock cycles are added. Memory and I/O read and write, as well as interrupt acknowledge cycles, can be extended by activating the WAIT input. For exact timing information, refer to the composite timing diagram.

Memory Read and Write Memory read and instruction fetch cycles are identical, except for the status information on the ST_0 – ST_3 outputs. During a memory read cycle, a 16-bit address is placed on the AD_0 – AD_{15} outputs early in the first clock

period, as shown in Figure 10. (In the Z8001, the 7-bit segment number is output on SN_0-SN_6 one clock period earlier than the 16-bit address offset to compensate for the delay in the memory management circuitry.)

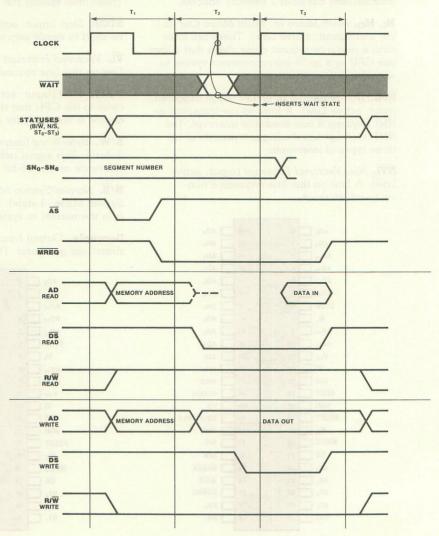


Figure 10. Memory Read and Write Timing

Memory Read and Write (Continued) A valid address is indicated by the rising edge of Address Strobe. Status and mode information become valid early in the memory access cycle and remain stable throughout. The state of the $\overline{\text{WAIT}}$ input is sampled in the middle of the second clock cycle by the falling edge of Clock. If $\overline{\text{WAIT}}$ is Low, an additional clock period is added between T_2 and T_3 . $\overline{\text{WAIT}}$ is sampled again in the middle of this wait cycle, and additional wait states can be inserted. This allows interfacing slow

memories. No control outputs change during wait states

Although Z8000 memory is word organized, memory is addressed as bytes. All instructions are word-aligned, using even addresses. Within a 16-bit word, the most significant byte (D_8 - D_{15}) is addressed by the low-order address (A_0 = Low), and the least significant byte (D_0 - D_7) is addressed by the high-order address (A_0 = High).

Input/ Output I/O timing is similar to memory read/write timing, except that one wait state is automatically inserted between T_2 and T_3 (Figure 11).

Both the segmented Z8001 and the non-segmented Z8002 use 16-bit I/O addresses.

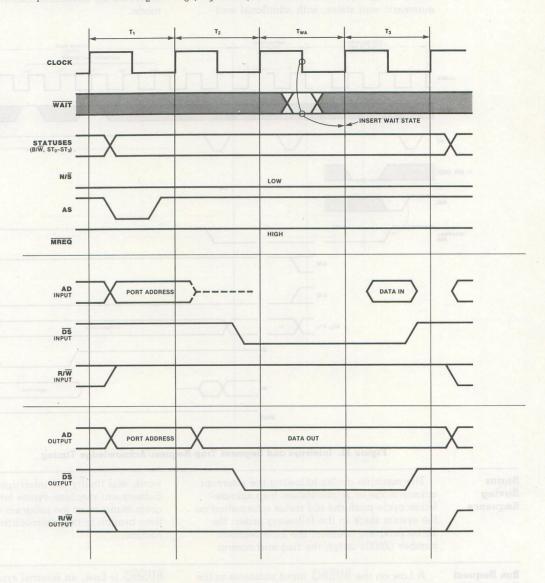


Figure 11. Input/Output Timing

Interrupt and Segment Trap Request and Acknowledge The Z8000 CPU recognizes three interrupt inputs (non-maskable, vectored and non-vectored) and a segmentation trap input. Any High-to-Low transition on the NMI input is asynchronously edge detected and sets the internal NMI latch. The VI, NVI and SEGT inputs as well as the state of the internal NMI latch are sampled at the beginning of T₃ in the last machine cycle of any instruction.

In response to an interrupt or trap, the subsequent IF₁ cycle is exercised, but aborted. The program counter is not updated, but the system stack pointer in decremented.

The next machine cycle is the interrupt acknowledge cycle. This cycle has five automatic wait states, with additional wait

states possible, as shown in Figure 12.

After the last wait state, the CPU reads the information on AD_0 - AD_{15} and stores it temporarily, to be saved on the stack later in the acknowledge sequence. This word identifies the source of the interrupt or trap. For the non-vectored and non-maskable interrupts, all 16 bits can represent peripheral device status information. For the vectored interrupt, the low byte is the jump vector, and the high byte can be extra user status. For the segmentation trap, the *high* byte is the Memory Management Unit identifier and the *low* byte is undefined.

After the acknowledge cycle, the $N\overline{S}$ output indicates the automatic change to system mode.

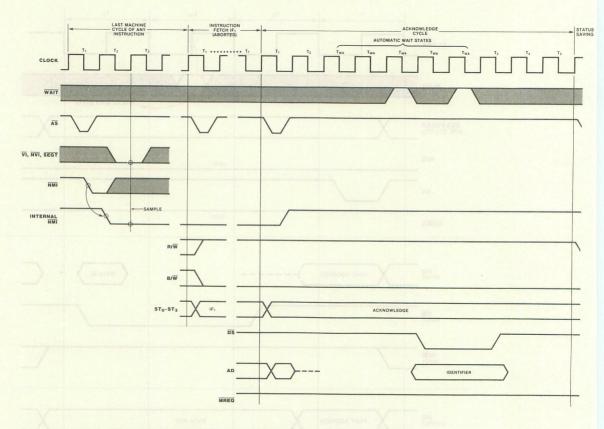


Figure 12. Interrupt and Segment Trap Request/Acknowledge Timing

Status Saving Sequence The machine cycles following the interrupt acknowledge or segmentation trap acknowledge cycle push the old status information on the system stack in the following order: the 16-bit program counter; the 7-bit segment number (Z8001 only); the flag and control

word; and finally the interrupt/trap identifier. Subsequent machine cycles fetch the new program status from the program status area, and then branch to the interrupt/trap service routine.

Bus Request Acknowledge Timing

A Low on the BUSRQ input indicates to the CPU that another device is requesting the Address/Data and Control buses. The asynchronous BUSRQ input is synchronized at the beginning of any machine cycle (Figure 13). If

BUSRQ is Low, an internal synchronous BUSRQ signal is generated, which-after completion of the current machine cycle-causes the BUSAK output to go Low and all bus outputs to go into the high-impedance state. The

Bus Request/ Acknowledge (Continued) requesting device-typically a DMA-can then control all buses.

When \overline{BUSRQ} is released, it is synchronized with the rising clock edge and the \overline{BUSAK} out-

put goes High one clock period later, indicating that the CPU will again take control of the

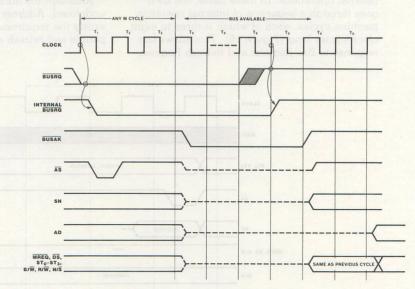


Figure 13. Bus Request/Acknowledge Timing

Stop

The \overline{STOP} input is sampled by the last falling clock edge immediately preceding any IF_1 cycle (Figure 14). If \overline{STOP} is found Low, a stream of memory refresh cycles is inserted after T_3 , again sampling the \overline{STOP} input on each falling clock edge in the middle of the T_3 states. This refresh operation does not use the

refresh prescaler or its divide-by-four clock prescaler; rather, it double-increments the refresh counter every three clock cycles. When \overline{STOP} is found High again, the next refresh cycle is completed, any remaining T states of the IF_1 cycle are then executed and the CPU continues its operation.

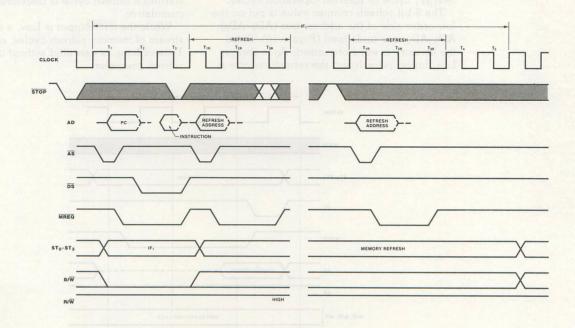


Figure 14. Stop Timing

Internal Operation Certain extended instructions such as Multiply and Divide, and some special instructions need additional time for the execution of internal operations. In these cases, the CPU goes through a sequence of internal operation machine cycles, each of which is three to eight clock cycles long (Figure 15). This allows fast response to Bus Request and Refresh Request.

because bus request or refresh cycles can be inserted at the end of any internal machine cycle.

Although the address outputs during T_1 are undefined, Address Strobe is generated to satisfy the requirements of future Z-Bus compatible self-refresh dynamic memories.

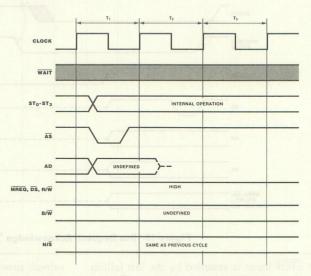


Figure 15. Internal Operation Timing

Memory Refresh When the 6-bit prescaler in the refresh counter has been decremented to zero, a refresh cycle consisting of three T-states is started as soon as possible (that is, after the next IF₁ cycle or Internal Operation cycle).

The 9-bit refresh counter value is put on the low-order side of the address bus (AD_0-AD_8) ; AD_9-AD_{15} are undefined (Figure 16). Since the memory is word-organized, A_0 is always Low during refresh and the refresh counter is

always incremented by two, thus stepping through 256 consecutive refresh addresses on AD_1 – AD_8 . Unless disabled, the presettable prescaler runs continuously and the delay in starting a refresh cycle is therefore not cumulative.

While the STOP input is Low, a continuous stream of memory refresh cycles, each three T-states long, is executed without using the refresh prescaler.

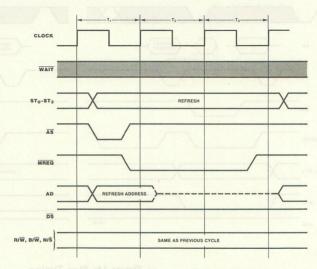


Figure 16. Memory Refresh Timing

Halt

A HALT instruction executes an unlimited number of 3-cycle internal operations, inter-spersed with memory refresh cycles whenever requested. An interrupt, segmentation trap or reset are the only exits from a HALT instruction. The CPU samples the \overline{VI} , \overline{NVI} , \overline{NMI} and \overline{SEGT} inputs at the beginning of every T_3 cycle. If an input is found active during two consecutive samples, the subsequent IF₁ cycle is exercised, but aborted, and the normal interrupt acknowledge cycle is started.

Reset

A Low on the \overline{RESET} input causes the following results within five clock cycles or less (Figure 17):

- □ AD₀-AD₁₅ are 3-stated
- □ AS, DS, MREQ,
 BUSAK and Mo are forced High
- □ ST₀-ST₃ and SN₀-SN₆ are forced Low
- □ Refresh is disabled
- \square R/W, B/W and N/S are not affected When RESET has been High for three clock

periods, two consecutive memory read cycles are executed in the system mode. In the Z8001, the first cycle reads the flag and control word from location 0002, the next reads the 7-bit program counter segment number from location 0004, the next reads the 16-bit PC offset from location 0006, and the following IF_1 cycle starts the program. In the Z8002, the first cycle reads the flag and control word from location 0002, the next reads the PC from location 0004 and the following IF_1 cycle starts the program.

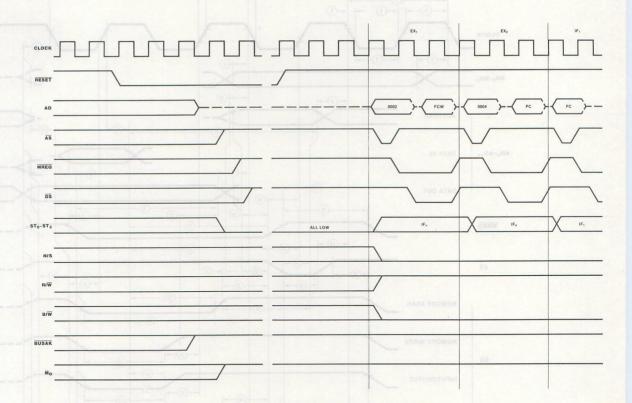
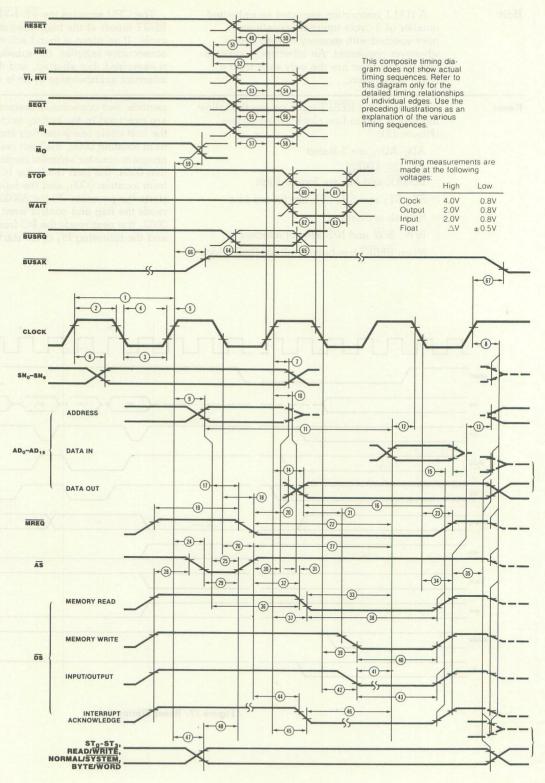


Figure 17. Reset Timing





| Telectrical 1 | | | | | | | |
|--|--|--------|--|----------------------------|----------------|----------------|--------|
| Characteristics | AC Electrical | Number | Symbol | Parameter | Min | Мах | Unit |
| 2 IwCh | | | | | 250 | 2000 | ns |
| Tic | e de la composition della comp | | | | 105 | 2000 | ns |
| Section | | | | | 105 | 2000 | ns |
| 6 TaC(SN) Clock I to Segment Number Valid (50 pF load) 7 TaC(SN) Clock I to Segment Number Not Valid 20 ns 18 8 TaC(SL) Clock I to Bus Float 100 ns 19 10 TaC(A) Clock I to Bus Float 100 ns 19 11 TaK(A) Clock I to Address Valid 100 ns 19 11 TaK(D) Date In to Clock I shaddress Float 11 TaK(D) Date In Required Valid 400 ns 19 12 TaD(C) Date In to Clock I Setup Time 70 ns 18 14 TaC(DO) Clock I to Date Out Valid 100 ns 16 16 TaD(SD) Date In to Clock I Setup Time 70 ns 18 16 TaC(MI) Date In to Electro I beld Time 0 ns 18 17 TaK(MI) Date In to Electro I beld Time 0 ns 18 18 TaC(MI) Clock I to MREQ I Delay 55 ns 18 18 TaC(MI) Clock I to MREQ I Delay 55 ns 18 19 TaC(MI) Clock I to MREQ I Delay 55 ns 18 10 TaC(MI) Clock I to MREQ I Delay 55 ns 18 10 TaC(MI) Clock I to MREQ I Delay 55 ns 18 11 TaK(MI) Clock I to MREQ I Delay 55 ns 18 12 TaC(MI) Clock I to MREQ I Delay 55 ns 18 13 TaC(MI) Clock I to MREQ I Delay 55 ns 18 14 TaC(MI) Clock I to MREQ I Delay 55 ns 18 15 TaC(MI) Clock I to MREQ I Delay 55 ns 18 16 TaC(MI) Clock I to MREQ I Delay 55 ns 18 17 TaK(D) MREQ I Delay 55 ns 18 18 TaC(MI) Clock I to MREQ I Delay 55 ns 18 18 TaC(MI) Clock I to MREQ I Delay 55 ns 18 18 TaC(MI) Clock I to MREQ I Delay 60 ns 18 18 TaC(MI) Clock I to MREQ I Delay 60 ns 18 18 TaC(MI) Clock I to MREQ I Delay 60 ns 18 18 TaC(MI) Clock I to MREQ I Delay 60 ns 18 18 TaC(MI) Clock I to MREQ I Delay 60 ns 18 18 TaC(MI) Clock I to MREQ I Delay 70 ns 18 18 TaC(MI) Clock I to MREQ I Delay 70 ns 18 18 TaC(MI) MREQ I Delay 70 ns 18 18 | | | | | | 20 | ns |
| 7 TGC(SNn) Clock 1 to Segment Number Not Valid 20 65 ns 9 TGC(A) Clock 1 to Address Valid 100 ns 9 TGC(A) Clock 1 to Address Valid 100 ns 110 TGC(A) Clock 1 to Address Valid 100 ns 121 TGC(A) Clock 1 to Address Float 11 TGA(DI) Address Valid to Data In Required Valid 400 65 ns 121 TGC(A) Clock 1 to Address Float 11 TGA(DI) Address Valid to Data In Required Valid 400 65 ns 121 TGC(A) DS 1 to Address Active 80 ns 121 TGC(DI) Clock 1 to Data Cut Valid 100 ns 121 TGC(DI) Clock 1 to Data Cut Valid 100 ns 121 TGC(DI) Clock 1 to Data Cut Valid 100 ns 121 TGC(DI) Clock 1 to Data Cut Valid 100 ns 121 TGC(DI) Clock 1 to Data Cut Valid 100 ns 121 TGC(MR) Clock 1 to MREQ 1 Delay 230 ns 121 TGC(MR) Clock 1 to MREQ 1 Delay 55 ns 121 TGC(MR) Clock 1 to MREQ 1 Delay 55 ns 121 TGC(MR) Clock 1 to MREQ 1 Delay 55 ns 121 TGC(MR) Clock 1 to MREQ 1 Delay 55 ns 121 TGC(MR) Clock 1 to MREQ 1 Delay 55 ns 121 TGC(MR) Clock 1 to MREQ 1 Delay 55 ns 121 TGC(MR) Clock 1 to MREQ 1 Delay 55 ns 121 TGC(MR) Clock 1 to MREQ 1 Delay 55 ns 121 TGC(MR) Clock 1 to MREQ 1 Delay 55 ns 121 TGC(MR) Clock 1 to MREQ 1 Delay 55 ns 122 TGC(MR) Clock 1 to MREQ 1 Delay 55 ns 122 TGC(MR) Clock 1 to MREQ 1 Delay 55 ns 122 TGC(MR) Clock 1 to MREQ 1 Delay 55 ns 122 TGC(MR) Clock 1 to MREQ 1 Delay 55 ns 122 TGC(MR) Clock 1 to MREQ 1 Delay 55 ns 122 TGC(MR) Clock 1 to MREQ 1 Delay 55 ns 123 TGC(MR) Clock 1 to MREQ 1 Delay 55 ns 123 TGC(MR) Clock 1 to MREQ 1 Delay 55 ns 123 TGC(MR) Clock 1 to MREQ 1 Delay 55 ns 123 TGC(MR) Clock 1 to MREQ 1 Delay 55 ns 123 TGC(MR) Clock 1 to MREQ 1 Delay 55 ns 123 TGC(MR) Clock 1 to MREQ 1 Delay 55 ns 123 TGC(MR) Clock 1 to MREQ 1 Delay 70 ns 123 TGC(MR) Clock 1 to MREQ 1 Delay 70 ns 123 TGC(MR) Clock 1 to MREQ 1 Delay 70 ns 123 TGC(MR) Clock 1 to MREQ 1 Delay 70 ns 123 TGC(MR) Clock 1 to MREQ 1 Delay 70 ns 123 TGC(MR) Clock 1 to MREQ 1 Delay 70 ns 124 TGC(MR) Clock 1 to MREQ 1 Delay 70 ns 124 TGC(MR) Clock 1 to MREQ 1 Delay 70 ns 124 TGC(MR) Clock 1 to MREQ 1 Delay 70 ns 124 TGC(MR) Clock 1 to MREQ 1 Delay 70 ns 124 TGC(MR) Clock 1 | | | | | | 20 — | ns |
| 8 TGC(E) Clock 1 to Bus Float 9 TGC(A) Clock 1 to Address Valid 1000 ms -10 — TGC(A) — Clock 1 to Address Valid 1000 ms -11 TdA(DI) Address Valid Data in Required Valid 400 11 TdA(DI) Address Valid Data in Required Valid 400 ms -12 TdD(C) Data in to Clock 1 Setup Time 70 ms -13 TdD(A) Data in to Clock 1 Setup Time 70 ms -14 TdC(DO) Clock 1 to Data Out Valid 100 ms -16 TdD(DS) Data in to DS 1 Hold Time 0 ms -16 TdD(DS) Data in to DS 1 Hold Time 0 ms -17 TdA(MR) Address Valid to MREO 1 Delay 55 ms -18 TdC(MR) Clock 1 to MREO 1 Delay 95 ms -18 TdC(MR) Clock 1 to MREO 1 Delay 96 ms -19 TdD(A) MREO 1 Delay 97 ms -20 — TdD(A) MREO 1 Delay 97 ms -21 TdD(DSW) MREO 1 Delay 100 ms -22 TdD(A) MREO 1 Delay 100 ms -23 TdC(MR) Clock 1 to MREO 1 Delay 98 ms -24 TdC(AS) Clock 1 to MREO 1 Delay 98 ms -25 — TdA(AS) — Address Valid to MREO 1 Delay 98 ms -26 TdC(AS) Clock 1 to MREO 1 Delay 98 ms -27 TdA(SI) — Address Valid to MREO 1 Delay 98 ms -28 TdA(AS) — Address Valid to AS 1 Delay 98 ms -29 TdA(AS) — Address Valid to AS 1 Delay 90 ms -29 TdA(AS) — Address Valid to AS 1 Delay 90 ms -29 TdA(AS) — Address Valid to AS 1 Delay 90 ms -29 TdA(SI) AS 1 to Data in Required Valid 290 ms -29 TdA(SI) AS 1 to Data in Required Valid 290 ms -29 TdA(SI) AS 1 to Data in Required Valid 290 ms -29 TdA(SI) AS 1 to Data in Required Valid 290 ms -29 TdA(SI) AS 1 to Data in Required Valid 90 ms -20 TdA(SI) AS 1 to Data in Required Valid 90 ms -20 TdA(SI) AS 1 to Data in Required Valid 90 ms -20 TdA(SI) AS 1 to Data in Required Valid 90 ms -21 TdA(DSI) AS 1 to Data in Required Valid 90 ms -22 TdA(DSI) AS 1 to Data in Required Valid 155 ms -23 TdA(DSI) AS 1 to Data in Required Valid 155 ms -24 TdC(DSI) Clock 1 to AS 1 Delay 90 ms -25 TdDS(DO) DS 1 to DS 1 to DS 1 to DS 1 Delay 90 ms -26 TdDS(DO) DS 1 to DS | | | | | | 130 | ns |
| 9 TdC(A) Clock to Address Valid 100 ms | | | | | 20 | | ns |
| Tack | | | the state of the s | | | 65 | ns |
| 11 | | | | | | 100 | ns |
| 12 Tablic() Data in to Clock Setup Time 70 ms | | | | | | 65 | ns |
| 13 TdDS(A) DS to Address Active 80 ms | | | | | | | ns |
| 14 TAC(DO) Clock It Date Out Valid 100 ns 161 171 | | | | | | | ns |
| Tab(DC) Date out Valid to DS Hold Time | | | | | 80 | | ns |
| 16 TaDO(DS) Data Out Valid to DS Delay 230 ns 17 TadA(MR) Address Valid to MREO Delay 5 ns 18 TaC(MR) Clock to MREO Delay 5 ns 19 TwMRh MREO With (High) 190 ns 20 TaMR(A) MREO Address Not Active 70 ns 21 TaDO(DSW) Data Out Valid to DS (Write) Delay 55 ns 22 TaMR(DI) MREO Lo Address Not Active 70 ns 23 TaC(MR) Clock to MREO Delay 80 ns 24 TaC(ASI) Clock to MREO Delay 80 ns 25 TadA(ASI) Address Valid to AS Delay 80 ns 26 TaC(ASI) Clock to MREO Delay 80 ns 27 TadAS(DI) AS to Data in Required Valid 290 ns 28 TaDS(AS) DS to AS Delay 70 ns 29 TwAS AS With (Low) 80 ns 30 TadAS(AS) AS to Address Not Active Delay 70 ns 31 TadAS(DSR) AS to DS (Read) Delay 70 ns 32 TadAS(DSR) AS to DS (Read) Delay 70 ns 33 TaDSR(DI) DS (Read) Delay 70 ns 34 TaC(DSR) Clock to DS (Delay) Delay 70 ns 35 TadS(DSR) AS to DS (Delay Delay 70 ns 36 TadA(DSR) Address Polat to DS (Delay) Delay 70 ns 36 TadA(DSR) Address Polat to DS (Delay) Delay 70 ns 36 TadA(DSR) Address Valid Delay 70 ns 37 TaC(DSR) DS (Read) Delay 70 ns 38 TwDSR DS (Read) Delay 70 ns 39 Tad(DSR) DS (Read) Delay 70 ns 30 Tad(DSR) DS (Read) Delay 70 ns 31 Tad(DSR) DS (Read) Delay 70 ns 32 Tad(DSR) DS (Read) Delay 70 ns 34 Tad(DSR) DS (Read) Delay 70 ns 35 Tad(DSR) DS (Read) Delay 70 ns 36 Tad(DSR) DS (Read) Delay 70 ns 37 Tad(DSR) DS (Read) Delay 70 ns 38 TwDSR DS (Read) Delay 70 ns 39 Tad(DSR) DS (Read) Delay 70 ns 40 TwDSW DS (Write) With (Low) 70 ns 41 Tad(DSR) DS (Ack.) to Data In Required Valid 15 ns 42 Tad(DSR) DS (Ack.) to Data In Required De | | | | | | 100 | ns |
| 17 | | | | | | | ns |
| 18 | | | | | | | ns |
| 19 | | | | | 55 | | ns |
| -20 | | | | Clock to MREQ Delay | | 80 | ns |
| 21 | | | | | | | ns |
| TadAR(DI) MREO I to Data In Required Valid 390 ns 74G(NAS) Clock I to MREO I Delay 80 ns 24 TdC(ASI) Clock I to MREO I Delay 80 ns 24 TdC(ASI) Clock I to MREO I Delay 90 ns 26 TdA(AS) Clock I to ASI Delay 90 ns 27 TdAS(DI) ASI I to Data In Required Valid 290 ns 28 TdAS(DI) ASI I to Data In Required Valid 290 ns 28 TdAS(DI) ASI I to Data In Required Valid 290 ns 29 TadS ASI Width (Low) 80 ns 70 ns 70 ns 70 70 70 70 70 70 70 7 | | | | | | | ns |
| 23 TaC(MR) Clock to MREQ Delay 80 ns | | | The second secon | | | | ns |
| 24 TaC(ASI) Clock to AS Delay 55 ns ns TaA(AS) Address Valid to AS Delay 90 ns 26 TaC(ASr) Clock to AS Delay 70 ns 27 TaAS(DI) AS to Data na Required Valid 290 ns 28 TaDS(AS) DS to AS Delay 70 ns 29 TaAS AS Width (Low) 80 ns 71 AS TabS(AS) AS to Address Not Active Delay 60 ns 71 AS TaAS(DS) AS to Address Not Active Delay 60 ns 71 AS TaAS(DSR) AS to Address Not Active Delay 70 ns 71 Address Not Active Delay 70 ns 71 AS TaAS(DSR) Address Float to DS (Read) Delay 70 ns 71 Address Float to DS (Read) Delay 70 ns 71 Address Float to DS (Read) Delay 70 ns 71 Address Float to DS (Read) Delay 70 ns 71 Address Float to DS (Read) Delay 70 ns 71 Address Float to DS (Read) Delay 70 ns 71 Address Float to DS (Read) Delay 70 ns 71 Address Float to DS (Read) Delay 70 ns 71 Address Float to DS (Read) Delay 70 ns 71 Address Float to DS (Read) Delay 70 ns 71 Address Float to DS (Read) Delay 70 ns 71 Address Float to DS (Read) Delay 70 ns 71 Address Float to DS (Read) Delay 70 ns 71 Address Float to DS (Read) Delay 70 ns 71 Address Float to DS (Read) Delay 70 ns 71 Address Float to DS (Read) Delay 70 ns 71 Address Float to DS (Read) Delay 70 ns 71 Address Float to DS (Read) Delay 70 ns 71 Address Float to DS (Read) Delay 71 Address Float Floa | | | | | 330 | 1 | ns |
| Address Valid to ĀS Delay | | | | | | | ns |
| 26 | | | The second secon | | | 80 | ns |
| 27 TdAS(DI) AS to Data In Required Valid 290 ns | | | | | 55 | | — ns — |
| 28 | | | | | | 90 | ns |
| 29 | | | | | | | ns |
| TidaS(A) | | | | | | | ns |
| 31 TdAs(DSR) Address Float to DS (Read) Delay 0 ns | | | | | | | ns |
| 32 TdAS(DSR) AS to DS (Read) Delay T0 ns 33 TdDSR(DI) DS (Read) Do late In Required Valid 155 ns 34 TdC(DSr) Clock to DS Delay T0 ns 36 TdA(DSR) Address Valid to DS (Read) Delay 120 ns 37 TdC(DSR) Clock to DS (Read) Delay 120 ns 38 TwDSR DS (Read) Delay 120 ns 39 TdC(DSW) Clock to DS (Read) Delay 120 ns 39 TdC(DSW) Clock to DS (Read) Delay 160 ns 40 TwDSW DS (Write) Delay 160 ns 41 TdDSI(DI) DS (Input) to Data In Required Valid 315 ns 42 TdC(DSR) Clock to DS (I/O) Delay 120 ns 43 TwDS DS (Write) Dista In Required Valid 315 ns 44 TdAS(DSA) AS to DS (Acknowledge) Delay 400 ns 44 TdAS(DSA) AS to DS (Acknowledge) Delay 960 ns 45 TdC(DSA) Clock to DS (Acknowledge) Delay 120 ns 46 TdDSA(DI) DS (Intus) DS (Acknowledge) Delay 420 ns 47 TdC(S) Clock to DS (Acknowledge) Delay 420 ns 48 TdS(AS) Status Valid to AS Delay 40 ns 49 TsR(C) RESET to Clock Setup Time 180 ns 51 TwNMI Nimit Clock Setup Time 180 ns 52 TsNMI(C) Nimit Clock Hold Time 0 ns 53 TsVI(C) VI, NVI to Clock Hold Time 0 ns 54 ThVI(C) VI, NVI to Clock Hold Time 0 ns 55 TsSGT(C) SEGT to Clock Setup Time 140 ns 56 ThSGT(C) SEGT to Clock Hold Time 0 ns 57 TsMI(C) MI to Clock Hold Time 0 ns 58 ThMI(C) MI to Clock Hold Time 0 ns 56 ThSTP(C) SEGT to Clock Hold Time 0 ns 57 TsMI(C) MI to Clock Hold Time 0 ns 58 ThMI(C) MI to Clock Hold Time 0 ns 59 TdC(MO) Clock Hold Time 0 ns 60 TdSTP(C) STOP to Clock Hold Time 0 ns 61 ThSTP(C) STOP to Clock Hold Time 0 ns 62 TdC(MSR(D) STOP to Clock Hold Time 0 ns 63 ThWI(C) Walt to Clock Hold Time 0 ns 66 TdC(BAR) | | | | | | | ns |
| 33 TdDSR(DI) DS (Read) to Data In Required Valid 155 ns 34 TdC(DST) Clock to DS Delay 70 ns 35 TdDS(DO) DS to Data Out and STATUS Not Valid 80 ns 36 TdA(DSR) Address Valid to DS (Read) Delay 120 ns 37 TdC(DSR) Clock to DS (Read) Delay 120 ns 38 TwDSR DS (Read) Width (Low) 275 ns 39 TdC(DSW) Clock to DS (Write) Delay 160 ns 40 TwDSW DS (Write) Width (Low) 160 ns 41 TdDSI(DI) DS (Write) Width (Low) 160 ns 42 TdC(DSI) Clock to DS (I/O) Delay 120 ns 43 TwDS DS (I/O) Width (Low) 400 ns 44 TdAS(DSA) AS to DS (Acknowledge) Delay 960 ns 45 TdC(DSA) Clock to DS (Acknowledge) Delay 960 ns 46 TdDSA(DI) DS (Acknowledge) Delay 960 ns 47 TdC(S) Clock to Data In Required Delay 420 ns 48 TdS(AS) Status Valid to AS Delay 400 ns 49 TsR(C) RESET to Clock Setup Time 180 ns 51 TwNMI NMI Width (Low) 100 ns 52 TsNMI(C) MI to Clock Setup Time 180 ns 54 ThVI(C) VI, NVI to Clock Setup Time 100 ns 55 TsSGT(C) SEGT to Clock Hold Time 0 ns 56 ThSGT(C) SEGT to Clock Hold Time 0 ns 57 TsMI(C) MI to Clock Setup Time 100 ns 58 ThMI(C) MI to Clock Setup Time 100 ns 59 TdC(MO) Clock Hold Time 0 ns 60 TsSTP(C) SEGT to Clock Hold Time 0 ns 61 ThSTP(C) STOP to Clock Hold Time 0 ns 62 TsWI(C) WAIT to Clock Setup Time 100 ns 63 ThWT(C) WAIT to Clock Setup Time 100 ns 64 TsBRQ(C) BUSRO to Clock Hold Time 0 ns 65 ThBRQ(C) BUSRO to Clock Hold Time 0 ns 66 TdC(BAK) Clock to BUSAK Delay 100 ns 67 TdC(BAK) Clock to BUSAK Delay 1 | | | | | | | ns |
| 34 | | | | | | | ns |
| TdDS(DO) | | | | | 155 | | ns |
| 36 | | | | | ord photo M in | 70 | ns |
| 37 TdC(DSR) Clock to \overline{DS} (Read) Delay 275 ns 38 TwDSR D\overline{DS} (Read) Width (Low) 275 ns 39 TdC(DSW) Clock to \overline{DS} (Write) Delay 95 ns -40 TwDSW D\overline{S} (Write) Width (Low) 160 ns -41 TdDSI(DI) D\overline{S} (Input) to Data In Required Valid 315 ns 42 TdC(DSf) Clock to \overline{DS} (I/O) Delay 120 ns 43 TwDS D\overline{S} (I/O) Width (Low) 400 ns 44 TdAS(DSA) AS to \overline{DS} (Acknowledge) Delay 960 ns -45 TdC(DSA) Clock to DS (Acknowledge) Delay 960 ns 46 TdDSA(DI) D\overline{S} (Acknowledge) Delay 420 ns 47 TdC(S) Clock to Data In Required Delay 420 ns 48 TdS(AS) Status Valid Delay 400 ns 49 TsR(C) RESET to Clock Setup Time 180 ns 49 TsR(C) RESET to Clock Setup Time 180 ns 51 TwNMI N\overline{MI} Width (Low) 100 ns 52 TsNMI(C) N\overline{MI} to Clock Setup Time 140 ns 53 TsVI(C) VI, N\overline{NVI} to Clock Setup Time 140 ns 54 ThVI(C) VI, N\overline{NVI} to Clock Setup Time 100 ns 55 TsSGT(C) SEGT to Clock Hold Time 0 ns 56 ThSGT(C) SEGT to Clock Hold Time 0 ns 57 TsMI(C) MI to Clock Setup Time 180 ns 58 ThMI(C) MI to Clock Setup Time 180 ns 59 TdC(MO) Clock to Mo Delay 120 ns 60 TsSTP(C) STOP to Clock Setup Time 140 ns 61 ThSTP(C) STOP to Clock Setup Time 0 ns 62 TsWT(C) WAIT to Clock Setup Time 0 ns 63 ThWT(C) WAIT to Clock Setup Time 0 ns 64 TsBRQ(C) BUSRQ to Clock Setup Time 0 ns 65 ThBRQ(C) BUSRQ to Clock Setup Time 0 ns 66 TdC(BAK) Clock to BUSAK Delay 100 ns 67 TdC(BAK) Clock to BUSAK Delay 100 ns 67 TdC(BAK) Clock to BUSAK Delay 100 ns 67 TdC(BAK) Clock Setup Time 100 ns 67 TdC(BAK) Clock Setu | | | | | | - COLOR | ns |
| 38 | | | | | 120 | | ns |
| TdC(DSW) Clock to \overline{DS} (Write) Delay 95 ns -40 | | | | | Lista ton | 120 | ns |
| TwDSW | | | | | 275 | | ns |
| All TdDSI(DI) DS (Input) to Data In Required Valid 315 ns | | | | | | 95 | ns |
| 120 ns | | | | | | | — ns — |
| A3 | | | | | 315 | | ns |
| AS to DS (Acknowledge) Delay 960 ns -45 | | | | | | 120 | ns |
| TdC(DSA) | | | | | | | ns |
| A6 | | | | | 960 | | ns |
| A7 | | | | | 100 | | ns |
| A8 | | | | | 420 | | ns |
| 49 TsR(C) RESET to Clock Setup Time 180 ns -50 | | | | | 40 | 110 | |
| Through Thro | | | | | | | |
| Signature Sign | | | | | | | ns |
| S2 | | | | | | | ns |
| TsVI(C) | | | | | | | ns |
| ThVI(C) | | | | Will to Clock Setup Time | | | ns |
| TsSGT(C) SEGT to Clock Setup Time 70 ns | | | | | | | ns |
| SEGT to Clock Hold Time 0 | | | | | | | ns |
| S7 | | | | | | | — ns — |
| S8 | | | | | | | ns |
| TdC(Mo) | | | | | | | |
| Test | | | | | 0 | 100 | |
| 61 ThSTP(C) STOP to Clock Hold Time 0 ns 62 TsWT(C) WAIT to Clock Setup Time 70 ns 63 ThWT(C) WAIT to Clock Hold Time 0 ns 64 TsBRQ(C) BUSRQ to Clock Setup Time 90 ns 65 ThBRQ(C) BUSRQ to Clock Hold Time 0 ns 66 TdC(BAKr) Clock to BUSRQ Delay 100 ns | | | | | 1.46 | 120 | ns |
| 62 TsWT(C) WAIT to Clock Setup Time 70 ns 63 ThWT(C) WAIT to Clock Hold Time 0 ns 64 TsBRQ(C) BUSRQ to Clock Setup Time 90 ns 65 ThBRQ(C) BUSRQ to Clock Hold Time 0 ns 66 TdC(BAKr) Clock to BUSRX Delay 100 ns | | | | | | | ns |
| 63 ThWT(C) WAIT to Clock Hold Time 0 ns 64 TsBRQ(C) BUSRQ to Clock Setup Time 90 ns 65 ThBRQ(C) BUSRQ to Clock Hold Time 0 ns 66 TdC(BAKr) Clock to BUSAK Delay 100 ns | | | | | | | ns |
| 64 TsBRQ(C) BUSRQ to Clock † Setup Time 90 ns 65 ThBRQ(C) BUSRQ to Clock † Hold Time 0 ns 66 TdC(BAKr) Clock † to BUSAK † Delay 100 ns | | | | | | | ns |
| -65 ThBRQ(C) BUSRQ to Clock † Hold Time 0 ns - 66 TdC(BAKr) Clock † to BUSAK † Delay 100 ns | | | | | | | ns |
| 66 TdC(BAKr) Clock t to BUSAK t Delay 100 ns | | | | | 90 | | ns |
| 67 TdC(BAVI) Clock the BUSAV Dalan | | | The second secon | | 0 | 100 | |
| of IdC(DANI) Clock I to DUSAN I Delay 100 ns | | | | | | | |
| | | 01 | IdC(DARI) | Clock I to DOSAK + Deldy | | 100 | ns |

Absolute Maximum Ratings

Voltages on all inputs and outputs
with respect to GND....-0.3 V to +7.0 V
Operating Ambient
Temperature.....0°C to +70°C
Storage Temperature....-65°C to +150°C

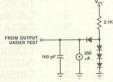
Stresses greater than those listed under Absolute Maximum Ratings may cause permanent damage to the device. This is a stress rating only; operation of the device at any condition above those indicated in the operational sections of these specifications is not implied. Exposure to absolute maximum rating conditions for extended periods may affect device reliability.

Standard Test Conditions

The characteristics below apply for the following standard test conditions, unless otherwise noted. All voltages are referenced to GND. Positive current flows into the referenced pin. Standard conditions are as follows:

$$\Box$$
 +4.75 V \leq V_{CC} \leq +5.25 V

$$\square$$
 0°C \leq T_A \leq +70°C



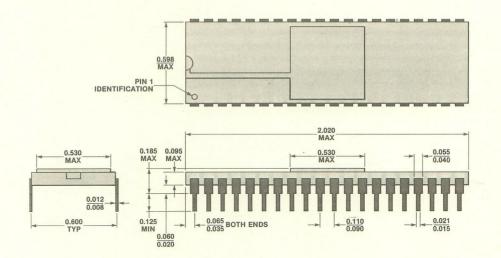
All ac parameters assume a load capacitance of 100 pF max, except for parameter 6 (50 pF max). Timing references between two output signals assume a load difference of 50 pF max.

| DC |
|------------|
| Character- |
| istics |

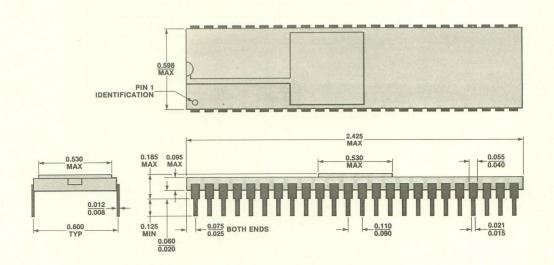
| Symbol | Parameter | Min | Мах | Unit | Condition |
|-----------------|--------------------------------|----------------------|-----------------------|------|--|
| V _{CH} | Clock Input High Voltage | V _{CC} -0.4 | V _{CC} + 0.3 | V | Driven by External Clock Generator |
| V _{CL} | Clock Input Low Voltage | -0.3 | 0.45 | V | Driven by External Clock Generator |
| V _{IH} | Input High Voltage | 2.0 | V _{CC} +0.3 | V | 22 IdMBI |
| V _{IL} | Input Low Voltage | -0.3 | 0.8 | V | A TONT |
| V _{OH} | Output High Voltage | 2.4 | Address Valu | V | $I_{OH} = -250 \mu\text{A}$ |
| V _{OL} | Output Low Voltage | I Delay | 0.4 | V | $I_{OL} = +2.0 \text{ mA}$ |
| I _{IL} | Input Leakage | Delay | ±10 | μΑ | $0.4 \le V_{\text{IN}} \le +2.4 \text{ V}$ |
| I _{OL} | Output Leakage | LW. | ±10 | μΑ | $0.4 \le V_{OUT} \le +2.4 \text{ V}$ |
| Icc | V _{CC} Supply Current | 18 20 St. | 300 | mA | DAAST IF |

| Ordering |
|-------------|
| Ordering |
| Information |

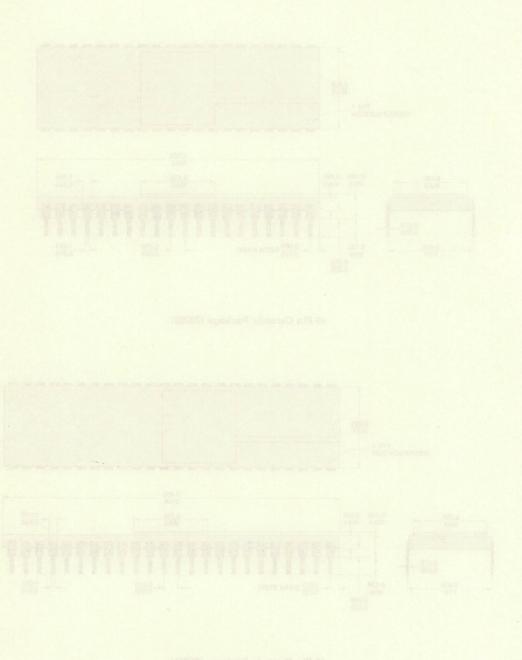
| Part Number | Temperature Range | Number of Pins | Package | Description | |
|-------------|----------------------|-------------------|---------|-------------------------------------|--|
| Z8001 CPU | 0°C to +70°C | 48 | Ceramic | Segmented 16-Bit Microprocessor | |
| Z8002 CPU | 0°C to +70°C | 40 | Ceramic | Non-Segmented 16-Bit Microprocessor | |



40-Pin Ceramic Package (Z8002)



48-Pin Ceramic Package (Z8001)



Z8010 MMU Memory Management Unit Product Specification



Product Specification

Preliminary

Features

- Dynamic segment relocation makes software addresses independent of physical memory addresses.
- Sophisticated memory-management features include access validation that protects memory areas from unauthorized or unintentional access, and a write-warning indicator that predicts stack overflow.
- 64 variable-sized segments from 256 to

- 65,536 bytes can be mapped into a total physical address space of 16 megabytes; all 64 segments are randomly accessible.
- Multiple MMUs can support several translation tables for each Z8001 address space.
- MMU architecture supports multi-programming systems and virtual memory implementations.

General Description

The Z8010 Memory Management Unit (MMU) manages the large 8M byte addressing spaces of the Z8001 CPU. The MMU provides dynamic segment relocation as well as numerous memory protection features.

Dynamic segment relocation makes user software addresses independent of the physical memory addresses, thereby freeing the user from specifying where information is actually located in the physical memory. It also provides a flexible, efficient method for supporting multi-programming systems. The MMU uses a translation table to transform the 23-bit logical address output from the Z8001 CPU into a 24-bit address for the physical memory. (Only logical memory addresses go to an MMU for translation; I/O addresses and data, in general, must bypass this component.)

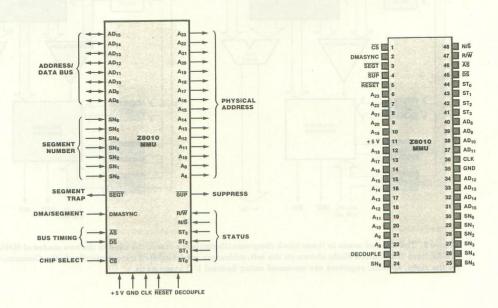


Figure 1. Pin Functions

Figure 2. Pin Assignments

General
Description
(Continued)

Memory segments are variable in size from 256 bytes to 64 kilobytes, in increments of 256 bytes. Pairs of MMUs support the 128 segment numbers available for the various Z8001 CPU address spaces. Within an address space, any number of MMUs can be used to accommodate multiple translation tables for System and Normal operating modes, or to support more sophisticated memory-management systems.

MMU memory-protection features safeguard memory areas from unauthorized or unintended access by associating special access restrictions with each segment. A segment is assigned a number of attributes when its descriptor initially entered into the MMU. When a memory reference is made, these attributes are checked against the status information supplied by the Z8001 CPU. If a

mismatch occurs, a trap is generated and the CPU is interrupted. The CPU can then check the status registers of the MMU to determine the cause.

Segments are protected by modes of permitted use, such as read only, system only, execute only and CPU-access only. Other segment management features include a writewarning zone useful for stack operations and status flags that record read or write accesses to each segment.

The MMU is controlled via 22 Special I/O instructions from the Z8001 CPU in System mode. With these instructions, system software can assign program segments to arbitrary memory locations, restrict the use of segments and monitor whether segments have been read or written.

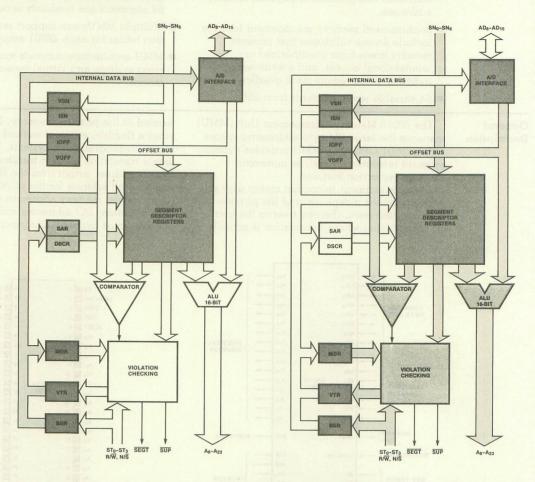


Figure 3. The shaded areas in these block diagrams illustrate the resources used in the two modes of MMU operation. In the Address Translation Mode shown on the left, addresses are translated automatically. In the Command Mode shown on the right, specific registers are accessed using Special I/O commands.

Segmented Addressing

A segmented addressing space—compared with linear addressing—is closer to the way a programmer uses memory because each procedure and data set can reside in its own segment.

The 8M byte Z8001 addressing spaces are divided into 128 relocatable segments of up to 64K bytes each. A 23-bit segmented address uses a 7-bit segment address to point to the segment, and a 16-bit offset to address any byte relative to the beginning of the segment. The two parts of the segmented address may be manipulated separately.

The MMU divides the physical memory into 256-byte blocks. Segments consist of physically contiguous blocks. Certain segments may be designated so that writes into the last block generate a warning trap. If such a segment is used as a stack, this warning can be used to increase the segment size and prevent a stack overflow error.

The addresses manipulated by the programmer, used by instructions and output by the Z8001 are called *logical addresses*. The MMU takes the logical addresses and transforms them into the *physical addresses* required for accessing the memory (Figure 4). This address transformation process is called *relocation*.

The relocation process is transparent to user software. A translation table in the MMU associates the 7-bit segment number with the base address of the physical memory segment.

The 16-bit logical address offset is added to the physical base address to obtain the actual physical memory location. Because a base address always has a low byte equal to zero, only the high-order 16 bits are stored in the MMU and used in the addition. Thus the low-order byte of the physical memory location is the same as the low-order byte of the logical address offset. This low-order byte therefore bypasses the MMU, thus reducing the number of pins required.

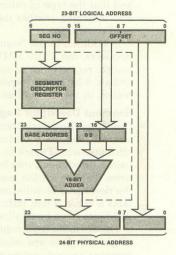


Figure 4. Logical-to-Physical Address Translation

Memory Protection

Each memory segment is assigned several attributes that are used to provide memory access protection. A memory request from the Z8001 CPU is accompanied by status information that indicates the attributes of the memory request. The MMU compares the memory request attributes with the segment attributes and generates a Trap Request whenever it detects an attribute violation. Trap Request informs the Z8001 CPU and the system control program of the violation so that appropriate action can be taken to recover. The MMU also generates the Suppress signal SUP in the event of an access violation. Suppress can be used by a memory system to inhibit stores into the memory and thus protect the contents of the memory from erroneous changes.

Five attributes can be associated with each segment. When an attempted access violates any one of the attributes associated with a segment, a Trap Request and a Suppress signal are generated by the MMU. These attributes are read only, execute only, system access only, inhibit CPU accesses and inhibit DMA accesses.

Segments are specified by a base address

and a range of legal offsets to this base address. On each access to a segment, the offset is checked against this range to insure that the access falls within the allowed range. If an access that lies outside the segment is attempted, Trap Request and Suppress are generated.

Normally the legal range of offsets within a segment is from 0 to 256N + 255 bytes, where 0≤N≤255. However, a segment may be specified so that legal offsets range from 256N to 65,535 bytes, where 0≤N≤255. The latter type of seament is useful for stacks since the Z8001 stack manipulation instructions cause stacks to grow toward lower memory locations. Thus when a stack grows to the limit of its allocated segment, additional memory can be allocated on the correct end of the segment. As an aid in maintaining stacks, the MMU detects when a write is performed to the lowest allocated 256 bytes of these segments and generates a Trap Request. No Suppress signal is generated so the write is allowed to proceed. This write warning can then be used to indicate that more memory should be allocated to the segment.

MMU Register Organization The MMU contains three types of registers: Segment Descriptor, Control and Status. A set of 64 Segment Descriptor Registers supplies the information needed to map logical memory addresses to physical memory locations. The segment number of a logical address determines which Segment Descriptor Register is used in address translation. Each Descriptor Register also contains the necessary information for checking that the segment location referenced is within the bounds of the segment and that the type of reference is permitted. It also indicates whether the segment has been read or written.

In addition to the Segment Descriptor Registers, the Z8010 MMU contains three 8-bit control registers for programming the device and six 8-bit status registers that record information in the event of an access violation.

Segment Descriptor Registers. Each of the 64 Descriptor Registers contains a 16-bit base address field, an 8-bit limit field and an 8-bit attribute field (Figure 5). The base address field is subdivided into high- and low-order bytes that are loaded one byte at a time when the descriptor is initialized. The limit field contains a value N that indicates N+1 blocks of 256 bytes have been allocated to the segment.*

The attribute field contains eight flags (Figure 6). Five are related to protecting the segment against certain types of access, one indicates the special structure of the segment, and two encode the types of accesses that have been made to the segment. A flag is set when its value is 1. The following brief descriptions indicate how these flags are used.

Read-Only (RD). When this flag is set, the segment is read only and is protected against any write access.

System-Only (SYS). When this flag is set, the segment can be accessed only in system mode, and is protected against any access in normal mode.

CPU-Inhibit (CPUI). When this flag is set, the segment is not accessible to the currently executing process, and is protected against any memory access by the CPU. The segment is, however, accessable under DMA.

Execute-Only (EXC). When this flag is set, the segment can be accessed only during an instruction fetch cycle, and thus is protected against any access during other cycles.

DMA-Inhibit (DMAI). When this flag is set, the segment can be accessed only by the CPU, and thus is protected against any access under DMA.

Direction and Warning (DIRW). When this flag is set, the segment memory locations are considered to be organized in descending order and each write to the segment is checked for access to the last 256-byte block. Such an access generates a trap to warn of potential segment overflow, but no Suppress signal is generated.

Changed (CHG). When this flag is set, the segment has been changed (written). This bit is set automatically during any write access to this segment if the write access does not cause any violation.

Referenced (REF). When this flag is set, the segment has been referenced (either read or written). This bit is set automatically during any access to the segment if the access does not cause a violation.



Figure 5. Segment Descriptor Registers

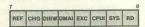


Figure 6. Attribute Field in Segment Descriptor Register

Control Registers. The three user-accessible 8-bit control registers in the MMU direct the functioning of the MMU (Figure 7). The Mode Register provides a sophisticated method for selectively enabling MMUs in multiple-MMU configurations. The Segment Address Register (SAR) selects a particular Segment Descriptor Register to be accessed during a control operation. The Descriptor Selection Counter Register points to a byte within the Segment Descriptor Register to be accessed during a control operation.



Figure 7. Control Registers

The Mode Register contains a 3-bit identification field (ID) that distinguishes among eight enabled MMUs in a multiple-MMU configuration. This field is used during the segment trap acknowledge sequence (refer to the section on Segment Trap and Acknowledge). In addition, the Mode Register contains five flags.

Multiple Segment Table (MST). This flag indicates whether multiple segment tables are present in the hardware configuration. When this flag is set, more than one table is present and the N/\overline{S} line must be used to determine whether the MMU contains the appropriate table.

Normal Mode Select (NMS). This flag indicates whether the MMU is to translate addresses when the N/\overline{S} line is High or Low. If the MST flag is set, the N/\overline{S} line must match the NMS flag for the MMU to translate segment addresses, otherwise the MMU Address lines remain 3-stated.

^{*}In the stack mode, segment size is 64K-256N.

MMU Register Organization (Continued) Upper Range Select (URS). This flag is used to indicate whether the MMU contains the lower-numbered segment descriptors or the higher-numbered segment descriptors. The most significant bit of the segment number must match the URS flag for the MMU to translate segment addresses, otherwise the MMU Address lines remain 3-stated.

Translate (TRNS). This flag indicates whether the MMU is to translate logical program addresses to physical memory locations or is to pass the logical addresses unchanged to the memory and without protection checking. In the non-translation mode, the most significant byte of the output is the 7-bit segment number and the most significant bit is 0. When this flag is set, the MMU performs address translation and attribute checking.

Master Enable (MSEN). This flag enables or disables the MMU from performing its address translation and memory protection functions. When this flag is set, the MMU performs these tasks; when the flag is clear the Address lines of the MMU remain 3-stated.

The Segment Address Register (SAR) points to one of the 64 segment descriptors. Control commands to the MMU that access segment descriptors implicitly use this pointer to select one of the descriptors. This register has an auto-incrementing capability so that multiple descriptors can be accessed in a block read/write fashion.

The Descriptor Selection Counter Register holds a 2-bit counter that indicates which byte in the descriptor is being accessed during the reading or writing operation. A value of zero in this counter indicates the high-order byte of the base address field is to be accessed, one indicates the low-order byte of the base address, two indicates the limit field and three indicates the attribute field.

Status Registers. Six 8-bit registers contain information useful in recovering from memory access violations (Figure 8). The Violation Type Register describes the conditions that generated the trap. The Violation Segment Number and Violation Offset Registers record the most-significant 15 bits of the logical address that causes a trap. The Instruction Segment Number and Offset Registers record the most-significant 15 bits of the logical address of the last instruction fetched before the first accessing violation. These two registers can be used in conjunction with external circuitry that records the low-order offset byte. At the time of the addressing violation, the Bus Cycle Status Register records the bus cycle status (status code, read/write mode and normal/system mode).

The MMU generates a Trap Request for two general reasons: either it detects an access

violation, such as an attempt to write into a read-only segment, or it detects a warning condition, which is a write into the lowest 256 bytes of a segment with the DIRW flag set. When a violation or warning condition is detected, the MMU generates a Trap Request and automatically sets the appropriate flags. The eight flags in the Violation Type Register describe the cause of a trap.

Read-Only Violation (RDV). Set when the CPU attempts to access a read-only segment and the R/\overline{W} line is Low.

System Violation (SYSV). Set when the CPU accesses a system-only segment and the $N\overline{/S}$ line is High.

CPU-Inhibit Violation (CPUIV). Set when the CPU attempts to access a segment with the CPU-inhibit flag set.

Execute-Only Violation (EXCV). Set when the CPU attempts to access an execute-only segment in other than an instruction fetch cycle.

Segment Length Violation (SLV). Set when an offset falls outside of the legal range of a segment.

Primary Write Warning (PWW). Set when an access is made to the lowest 256 bytes of a segment with the DIRW flag set.

Secondary Write Warning (SWW). Set when the CPU pushes data into the last 256 bytes of a system stack and EXCV, CPUIV, SLV, SYSV, RDV or PWW is set. Once this flag is set, subsequent write warnings for accessing the system stack do not generate a Segment Trap request.

Fatal Condition (FATL). Set when any other flag in the Violation Type Register is set and either a violation is detected or a write warning condition occurs in normal mode. This flag is not set during a stack push in system mode that results in a warning condition. This flag indicates a memory access error has occurred in the trap processing routine. Once set, no Trap Request signals are generated on subsequent violations. However, Suppress signals are generated on this and subsequent CPU violations until the FATL flag has been reset.

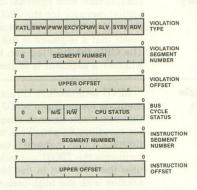


Figure 8. Status Registers

Segment Trap and Acknowledge

The Z8010 MMU generates a Segment Trap when it detects an access violation or a write warning condition. In the case of an access violation, the MMU also activates Suppress, which can be used to inhibit memory writes and to flag special data to be returned on a read access. Segment Trap remains Low until a Trap Acknowledge signal is received. If a CPU-generated violation occurs, Suppress is asserted for that cycle and all subsequent CPU instruction execution cycles until the end of the instruction. Intervening DMA cycles are not suppressed, however, unless they generate a violation. Violations detected during DMA cycles cause Suppress to be asserted during that cycle only-no Segment Trap Requests are ever generated during DMA cycles.

Segment traps to the Z8001 CPU are handled similarly to other types of interrupts. To service a segment trap, the CPU issues a segment trap acknowledge cycle. The acknowledge cycle is always preceded by an instruction fetch cycle that is aborted (the MMU has been designed so that this dummy cycle is ignored). During the acknowledge cycle all enabled MMUs use the Address/Data lines to indicate their status. An MMU that has generated a Segment Trap Request outputs a 1 on the A/D line associated with the number in its ID field; an MMU that has not generated a segment trap request outputs a 0 on its associated A/D line. A/D lines for which no MMU is associated remain 3-stated. During a

segment trap acknowledge cycle, an MMU uses A/D line 8+i if its ID field is i.

Following the acknowledge cycle the CPU automatically pushes the Program Status and Program Counter onto the system stack and loads another Program Status and Program Counter from the Program Status Area. The Segment Trap line is reset during the segment trap acknowledge cycle. Suppress is not generated during the stack push. If the store creates a write warning condition, a Segment Trap Request is generated and is serviced at the end of the context swap. The SWW flag is also set. Servicing this second Segment Trap Request also creates a write warning condition. but because the SWW flag is set, no Segment Trap Request is generated. If a violation rather than a write warning occurs during the context swap, the FATL flag is set rather than the SWW flag. Subsequent violations cause Suppress to be asserted but not Segment Trap Request. Without the SWW and FATL flags. trap processing routines that generate memory violations would repeatedly be interrupted and called to process the trap they created.

The CPU routine to process a trap request should first check the FATL flag to determine if a fatal system error has occurred. If not, the SWW flag should be checked to determine if more memory is required for the system stack. Finally, the trap itself should be processed and the Violation Type Register reset.

Virtual Memory

Several features of the MMU can be used in conjunction with external circuitry to support virtual memory for the Z8001. Segment Trap Request can be used to signal the CPU in the event that a segment is not in primary memory. The CPU-Inhibit Flag can be used to indicate whether a segment is in the memory or in

secondary storage. The Changed and Altered Flags in the attribute field for each segment can aid in implementing efficient segment management policies. The Status Registers can be used in recovering from virtual memory access faults.

Multiple MMUs

MMU architecture directly supports two methods for multiple MMU configurations. The first approach extends single-MMU capability for handling 64 segments to a dual-MMU configuration that manages the 128 different segments the Z8001 can address. This scheme uses the URS flag in the Mode Register in connection with the high-order bit of the segment number (SN_6).

The second approach uses several MMUs to implement multiple translation tables. Multiple tables can be used to reduce the time required to switch tasks by assigning separate tables to each task. Multiple translation tables for multi-

task environments can use the Master Enable Flag to enable the appropriate MMUs through software. Multiple translation tables may also be used to extend the physical memory size beyond 16 megabytes by separating system from normal memory and/or program from data memory. The MST and NMS flags in the Mode Register can be used in conjunction with the $\overline{\rm N/S}$ line to select the MMU that contains the appropriate table. Special external circuitry that monitors the CPU Status lines can manipulate the MMU $\overline{\rm N/S}$ line to perform this selection.

DMA Operation

Direct memory access operations may occur between Z8001 instruction cycles and can be handled through the MMU. The MMU permits DMA in either the System or Normal mode of operation. For each memory access, the segment attributes are checked and if a violation is detected, Suppress is activated. Unlike a CPU violation that automatically causes Suppress signals to be generated on subsequent memory accesses until the next instruction, DMA violations generate a Suppress only on a per memory access basis.

The DMA device should note the Suppress signal and record sufficient information to enable the system to recover from the access violation. No Segment Trap Request is ever generated during DMA, hence warning conditions are not signaled. Trap Requests are not issued because the CPU cannot acknowledge such a request.

At the start of a DMA cycle, DMASYNC must go Low, indicating to the MMU the beginning of a DMA cycle. A Low DMASYNC inhibits the MMU from using an indeterminate segment number on lines SNo-SN6. When the DMA logical memory address is valid, the DMASYNC line must be High on a rising edge of Clock and the MMU then performs its address translation and access protection functions. Upon the release of the bus at the termination of the DMA cycle the DMASYNC line must again be High. After two clock cycles of DMASYNC High, the MMU assumes that the CPU has control of the bus and that subsequent memory references are CPU accesses. The first instruction fetch occurs at least two cycles after the CPU regains control of the bus. During CPU cycles, DMASYNC should always be High.

MMU Commands

The various registers in the MMU can be read and written using Z8001 CPU special I/O commands. These commands have machine cycles that cause the Status lines to indicate an SIO operation is in progress. During these machine cycles the MMU enters command mode. In this mode, the rising edge of the Address Strobe indicates a command is present on the AD_8 - AD_{15} . If this command indicates that data is to be written into one of the MMU registers, the data is read from AD_8 - AD_{15} while Data Strobe is Low. If the command indicates that data is to be read from one of the MMU registers, the data is placed on AD_8 - AD_{15} while Data Strobe is Low.

There are ten commands that read or write various fields in the Segment Descriptor Register. The status of the Read/Write line indicates whether the command is a read or a write

The auto-incrementing feature of the Segment Address Register (SAR) can be used to block load segment descriptors using the repeat forms of the Special I/O instructions. The SAR is autoincremented at the end of the field. In accessing the base field, first the high-order byte is selected and then the low-order byte. The command accessing the entire Descriptor Register references the fields in the order of base address, limit and attribute.

| Opcode (Hex) | Instruction |
|--------------|---|
| 08 | Read/Write Base Field |
| 09 | Read/Write Limit Field |
| 0A | Read/Write Attribute Field |
| OB | Read/Write Descriptor (all fields) |
| 0C | Read/Write Base Field; Increment SAR |
| 0D | Read/Write Limit Field; Increment SAR |
| OE | Read/Write Attribute Field; Increment SAR |
| OF | Read/Write Descriptor; Increment SAR |
| 15 | Set All CPU-Inhibit Attribute Flags |
| 16 | Set All DMA-Inhibit Attribute Flags |

Three commands are used to read and write the control registers.

| Opcode (Hex) | Instruction |
|--------------|--|
| 00 | Read/Write Mode Register |
| 01 | Read/Write Segment Address Register |
| 20 | Read/Write Descriptor Selector Counter |

The Status Registers are read-only registers, although the Violation Type Register (VTR) can be reset. Nine instructions access these registers.

| Opcode (Hex) | Instruction |
|--------------|---|
| 02 | Read Violation Type Register |
| 03 | Read Violation Segment Number Register |
| 04 | Read Violation Offset (High-byte) Register |
| 05 | Read Bus Status Register |
| 06 | Read Instruction Segment Number Register |
| 07 | Read Instruction Offset (High-byte) Register |
| 11 | Reset Violation Type Register |
| 13 | Reset SWW Flag in VTR |
| 14 | Reset FATL Flag in VTR |

MMU Timing The Z8010 translates addresses and checks for access violations by stepping through sequences of basic clock cycles corresponding to the cycle structure of the Z8001 CPU. The following timing diagrams show the relative timing relationships of MMU signals during the basic operations of memory read/write and MMU control commands. For exact timing information, refer to the composite timing diagram.

Memory Read and Write. Memory read and instruction fetch cycles are identical, except for the status information on the ST_0 – ST_3 inputs. During a memory read cycle (Figure 9) the 7-bit segment number is input on SN_0 – SN_6 one clock period earlier than the address offset; a High on DMASYNC during T_3 indicates that the segment offset data is valid. The most significant eight bits of the address offset are placed on the AD_0 – AD_{15} inputs early in the

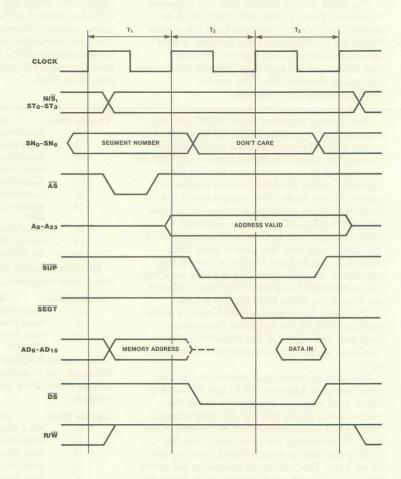


Figure 9. Memory Read Timing

MMU
Timing
(Continued)

first clock period. Valid address offset data is indicated by the rising edge of Address Strobe. Status and mode information become valid early in the memory access cycle and remain stable throughout. The most significant 16-bits of the address (physical memory location) remain valid until the end of T₃. Segment Trap Request and Suppress are asserted in T₂.

Segment Trap Request remains Low until Segment Trap Acknowledge is received. Suppress is asserted during the current machine cycle and terminates during T₃. Suppress is repeatedly asserted during CPU instruction execution cycles until the current instruction has terminated.

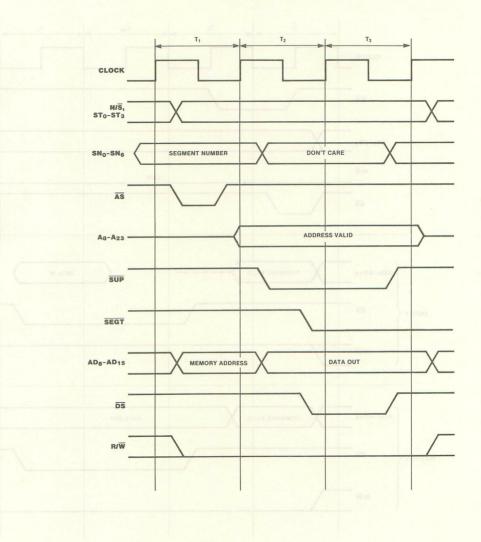


Figure 10. Memory Write Timing

MMU Timing (Continued) **MMU Command Cycle.** During the command cycle of the MMU (Figure 11), commands are placed on the Address/Data lines during T₁. The Status lines indicate that a special I/O instruction is in progress, and the Chip Select line enables the appropriate MMU for that command. Data to be written to a register in the MMU must be valid on the Address/Data lines late in T₂. Data read from the MMU is

placed on the Address/Data lines late in the $T_{\rm WA}$ cycle.

Input/Output and Refresh. Input/Output and Refresh operations are indicated by the status lines ST_0 – ST_3 . During these operations, the MMU refrains from any address translation or protection checking. The address lines A_8 – A_{15} remain 3-stated.

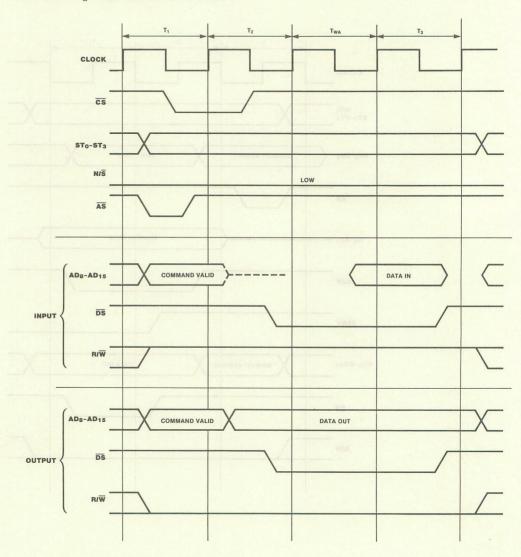


Figure 11. I/O Command Timing

MMU
Timing
(Continued)

Reset. The MMU can be reset by either hardware or software mechanisms. A hardware reset occurs on the falling edge of the Reset signal; a software reset is performed by a Z8001 special I/O command. A hardware reset clears the Mode Register, Violation Type Register and Descriptor Selection Counter. If the Chip Select line is Low, the Master Enable Flag in the Mode Register is set to 1. All other registers are undefined. After reset, the AD₈-AD₁₅ and A₈-A₂₃ lines are 3-stated. The SUPand SEGT open-drain outputs are not driven. If the Master Enable flag is not set during reset, the MMU does not respond to subsequent addresses on its A/D lines. To enable an MMU after a hardware reset, an MMU command must be used in conjunction with the Chip Select line.

A software reset occurs when the Reset Violation Type Register command is issued. This command clears the Violation Type Register and returns the MMU to its initial state (as if no violations or warnings had occurred). Note that the hardware and software resets have different effects

Segment Trap and Acknowledge. The Z8010 MMU generates a segment trap whenever it detects an access violation or a write into the lowest block of a segment with the DIRW flag set. In the case of an access violation, the

MMU also activates Suppress. This Suppress signal can be used to inhibit memory writes and to flag special data to be returned on a read access. The Segment Trap remains Low until a Trap Acknowledge signal is received. If a violation occurs, Suppress is asserted for that cycle and all subsequent CPU cycles until the end of the instruction; intervening DMA cycles are not suppressed, however, unless they generate a violation. Violations detected during DMA cycles cause Suppress to be asserted during that cycle only, but no Trap Request is generated.

When the MMU issues a Segment Trap Request it awaits a Segment Trap Acknowledae. Subsequent violations occurring before the Trap Acknowledge is received are still detected and handled appropriately. During the Segment Trap Acknowledge cycle, the MMU drives one of its Address/Data lines High; the particular line selected is a function of the identification field of the mode register. After the Segment Trap has been acknowledged by the Z8001 CPU, the Violation Status Register should be read via the Special I/O commands in order to determine the cause of the trap. The Trap Type Register should also be reset so that subsequent traps will be recorded correctly.

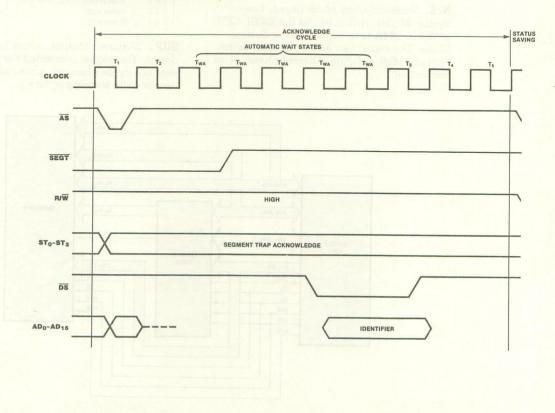


Figure 12. Segment Trap and Acknowledge Timing

Pin Description

A₈-A₂₃. Address Bus (outputs, active High, 3-state). These address lines are the 16 most-significant bits of the physical memory location

AD₈-AD₁₅. Address/Data Bus (inputs/outputs, active High, 3-state). These multiplexed address and data lines are used both for commands and for logical addresses intended for translation.

 $\overline{\text{AS}}$. Address Strobe (input, active Low). The rising edge of $\overline{\text{AS}}$ indicates that AD_0 - AD_{15} , ST_0 - ST_3 , R/\overline{W} and N/\overline{S} are valid.

CLK. System Clock (input). CLK is the 5 V single-phase time-base input used for both the CPU and MMU.

CS. Chip Select (input, active Low). This line selects an MMU for a control command.

Decouple. Output from on-chip negative substrate-bias generator.

DMASYNC. DMA/Segment Number Synchronization Strobe (input, active High). A Low on this line indicates a DMA access is occurring; a High indicates the segment number is valid. It must always be High during CPU cycles.

DS. Data Strobe (input, active Low). This line provides timing for the data transfer between the MMU and the Z8001 CPU.

N/S. Normal/System Mode (input, Low = System Mode). N/S indicates the Z8001 CPU or Z8016 DMA is in the Normal or System Mode. The signal can also be used to switch between MMUs during different phases of an instruction.

RESET. Reset (input, active Low). A Low on this line resets the MMU.

 R/\overline{W} . Read/Write (input, Low = write). R/\overline{W} indicates the Z8001 CPU or Z8016 DMA is reading from or writing to memory or the MMU.

SEGT. Segment Trap Request (output, active Low, open drain). The MMU interrupts the Z8001 CPU with a Low on this line when the MMU detects an access violation or write warning.

 SN_0 - SN_6 . Segment Number (inputs, active High). The SN_0 - SN_5 lines are used to address one of 64 segments in the MMU; SN_6 is used to selectively enable the MMU.

ST₀-ST₃. Status (inputs, active High). These lines specify the Z8001 CPU status.

| ST3-ST0 | Definition |
|---------|---|
| 0000 | Internal operation |
| 0001 | Memory refresh |
| 0010 | I/O reference |
| 0011 | Special I/O reference (e.g., to an MMU) |
| 0100 | Segment trap acknowledge |
| 0101 | Non-maskable interrupt acknowledge |
| 0110 | Non-vectored interrupt acknowledge |
| 0111 | Vectored interrupt acknowledge |
| 1000 | Data memory request |
| 1001 | Stack memory request |
| 1010 | Reserved |
| 1011 | Reserved |
| 1100 | Instruction space access |
| 1101 | Instruction fetch, first word |
| 1110 | Reserved |
| 1111 | Reserved |
| | |

SUP. Suppress (output, active Low, open drain). This signal is asserted during the current bus cycle when any access violation except write warning occurs.

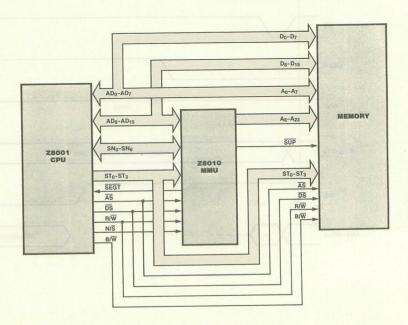


Figure 13. The MMU in a Z8001 System

| Absolute |
|----------|
| Maximum |
| Ratinas |

Voltages on all inputs and outputs
with respect to GND....-0.3 V to +7.0 V

Operating Ambient
Temperature....0°C to + 70°C

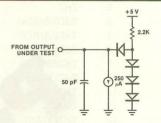
Storage Temperature...-65°C to +150°C

Stresses greater than those listed under Absolute Maximum Ratings may cause permanent damage to the device. This is a stress rating only; operation of the device at any condition above those indicated in the operational sections of these specifications is not implied. Exposure to absolute maximum rating conditions for extended periods may affect device reliability.

Standard Test Conditions

The characteristics below apply for the following standard test conditions, unless otherwise noted. All voltages are referenced to GND. Positive current flows into the referenced pin. Standard conditions are as follows:

- $+4.75 \text{ V} \le \text{V}_{CC} \le +5.25 \text{ V}$
- GND = 0 V
- $0^{\circ}C \leq T_{A} \leq +70^{\circ}C$



DC Characteristics

| Symbol | Parameter | Min | Мах | Unit | Condition |
|-----------------|--------------------------------|----------------------|----------------------|------|-------------------------------------|
| V _{CH} | Clock Input High Voltage | V _{CC} -0.4 | V _{CC} +0.3 | V | Driven by External Clock Generator |
| V _{CL} | Clock Input Low Voltage | -0.3 | 0.45 | V | Driven by External Clock Generator |
| V _{IH} | Input High Voltage | 2.0 | V _{CC} +0.3 | V | COMPAT 81 |
| V _{IL} | Input Low Voltage | -0.3 | 0.8 | V | nusjum ei |
| V _{OH} | Output High Voltage | 2.4 | NEW YEAR | V | $I_{OH} = -250 \mu\text{A}$ |
| V _{OL} | Output Low Voltage | | 0.4 | V | $I_{OL} = +2.0 \text{ mA}$ |
| I _{IL} | Input Leakage | - al brisk (| ±10 | μΑ | $0.4 \le V_{IN} \le +2.4 \text{ V}$ |
| I _{OL} | Output Leakage | G bifaV jok | ±10 | μΑ | $0.4 \le V_{IN} \le +2.4 \text{ V}$ |
| I _{CC} | V _{CC} Supply Current | Address Or | 300 | mA | APPROME IS |

| Ordering | |
|-----------|----|
| Informati | 01 |

| Part Number | Temperature Range | Number of Pins | Package | Description | | |
|-------------|----------------------|-------------------|---------|------------------------|--|--|
| Z8010 MMU | 0°C to +70°C | 48 | Ceramic | Memory Management Unit | | |

| Character- istics | 1 | TcC | Clock Cycle Time | 250 | | |
|----------------------|----|------------|--|------|-----|--------|
| | 2 | TwCh | Clock Width (High) | 105 | | |
| | 3 | TwCl | Clock Width (Low) | 105 | | |
| | 4 | TfC | Clock Fall Time | | 25 | |
| | 5 | TrC | Clock Rise Time | | 25 | |
| | 6 | TdDSA(RDv) | DS (Acknowledge) to Read Data Valid Delay | | 100 | [1] |
| | 7 | TdDSA(RDf) | DS † (Acknowledge) to Read Data Float Delay | 20 | 75 | [1] |
| | 8 | TdDSR(RDv) | DS ↓ (Read) to AD Output Driven Delay | | 100 | [1] |
| | 9 | TdDSR(RD() | DS † (Read) to Read Data Float Delay | 20 | 75 | [1] |
| | 10 | TdC(WDv) | CLK † to Write Data Valid Delay | | 160 | |
| | 11 | The (WDn) | CLK to Write Data Not Valid Hold Time | 30 | | |
| | 12 | TwAS | Address Strobe Width | 60 | | |
| | 13 | TsOFF(AS) | Offset Valid to AS † Setup Time | 45 | | |
| | 14 | ThAS(OFFA) | S 1 to Offset Not Valid Hold Time | 60 | | |
| | 15 | TdAS(C) | AS La CLK Delay | 110 | | |
| | 16 | TdDS(AS) | DS No AS I Delay | 50 | | |
| | 17 | TdAS(DS) | AS 1 to DS Dolay | 50 | | |
| | 18 | TsSN(C) | SN Data Valid to CLK † Setup Time | 120 | | |
| | 19 | ThC(SNn) | CLK to SN Data No. Walid Hold Time | 0 | | |
| | 20 | TdDMAS(C) | DMASYNC Valid to CLK L Delay | 120 | | |
| | 21 | TdSTNR(AS) | Status (ST ₀ -ST ₃ , N/S, R/W) Valid to AS † Delay | 60 | | |
| | 22 | TdC(DMA) | Clk 1 to DMASYNC 1 Delay | 20 | | |
| | 23 | TdST(C) | Status (ST ₀ -ST ₃) Valid to CLK 1 Delay | 140 | | |
| | 24 | TdDS(STn) | DS † to Status Not Valid Delay | 0 | | |
| | 25 | TdOFF(Av) | Offset Valid to Address Output Valid Delay | | 175 | [1,4, |
| | 26 | TdST(Ad) | Status Valid to Address Output Driven Delay | | 155 | [1,4,6 |
| | 27 | TdDS(Af) | DS † to Address Output Float Delay | 30 | 160 | [1] |
| | 28 | TdAS(Ad) | AS I to Address Output Driven Delay | 2 | 145 | [1, 4 |
| | 29 | TdC(Av) | CLK † to Address Output Valid Delay | TAN | 255 | [1, 4 |
| | 30 | TdAS(SEGT) | AS t to SEGT Delay | | 160 | [1, 2 |
| | 31 | TdC(SEGT) | CLK † to SEGT † Delay | | 300 | [1, 2 |
| | 32 | TdAS(SUP) | AS † to SUP Delay | | 150 | [1, 2 |
| | 33 | TdDS(SUP) | DS t to SUP t Delay | 30 | 155 | [1, 2 |
| | 34 | TsCS(AS) | Chip Select Input Valid to AS † Setup Time | 10 | | |
| | 35 | ThAS(CSn) | AS † to Chip Select Input Not Valid Hold Time | 80 | | |
| | 36 | TdAS(C) | AS † to CLK † Delay | 0 | | |
| | 37 | TsCS(RST) | Chip Select Input Valid to RESET † Setup Time | 150 | | |
| | 38 | ThRST(CSn) | RESET † to Chip Select Input Not Valid Hold Time | 0 | | |
| | 39 | TwRST | RESET Width (Low) | 2TcC | | |
| | 40 | TdC(RDv) | CLK † to Read Data Valid Delay | | 460 | [1] |
| | 41 | TdDS(C) | DS † to CLK † Delay | 30 | | |
| | 42 | TdC(DS) | CLK to DS Delay | 0 | 110 | |

^{[1] 50} pF Load.

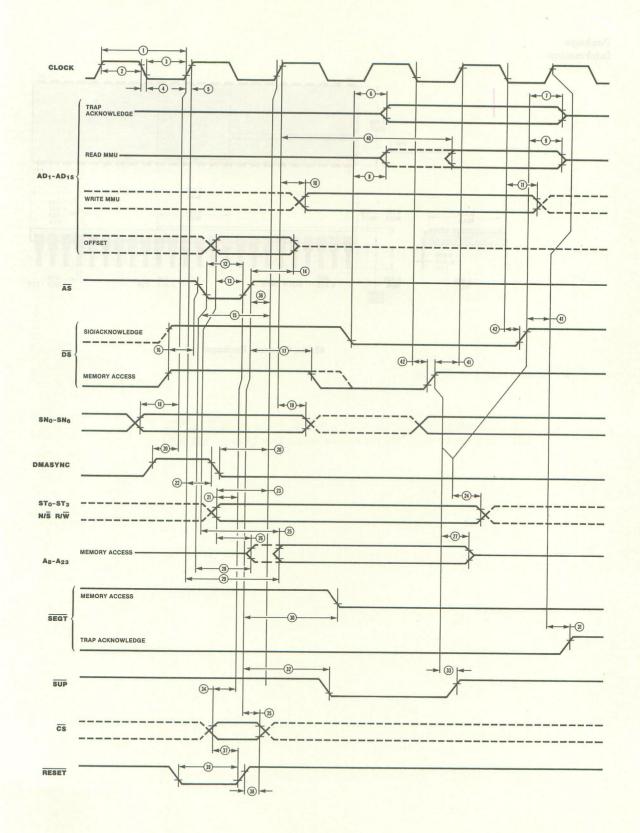
Product Specification is specified at 100 ns with 100 pF load. With a load of 50 pF, this delay is reduced to 80 ns. [6] Parameter 47 (Clock 1 to Status Valid Delay) in the Z8001/2 CPU Product Specification is specified at 110 ns with a 100 pF load. With a 50 pF load, this delay is reduced to 100 ns.

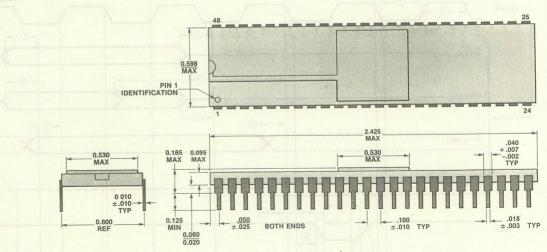
^{[2] 2.2}K Pull-up.

^[3] All values in nanoseconds.

^[4] These values apply to the Z8010-3 version only. A Z8010-2 version available soon will improve these values by 40 ns.

^[5] Parameter 9 (Clock 1 to Address Valid) in the Z8001/2 CPU





48-Pin Ceramic Package

A Small Z8000 System Application Note



Application Note

Peter Alfke

Introduction

This application note describes the complete hardware design implementation of a small computer using the Zilog Z8002 16-bit microprocessor, ROMs/EPROMs and dynamic RAMs plus parallel and serial I/O devices. The interface requirements of the Z8002 to memory and to Z-80 peripherals are described and design alternatives are given whenever possible.

This design is similar in structure and is software compatible with the Zilog Z8000 Development Module (part number 05-6101-01), a more elaborate design that offers additional features, such as strapping options for I/O space, choice of 4K or 16K RAMs, 2.5 and 4 MHz operation from the same 20 MHz oscillator.

This application note describes a simple and small practical Z8000 system using a minimal number of TTL support devices. Whenever possible, gate functions have been combined into MSI circuits to achieve lowest component count and thus the lowest manufacturing cost,

not necessarily the lowest component cost. The result is a tight design that uses MSI TTL circuits in a very efficient—but sometimes non-obvious—way that minimizes the package count. Because some of the design tricks may not be self-explanatory, an effort has been made to explain them.

A total of 22 TTL packages are used to control the 23 to 37 MOS-LSI logic and memory devices. Seven TTL packages generate the CPU clock and buffer addresses, data and control lines and decode Status. Ten TTL packages, latch the addresses, multiplex them into dynamic RAMs, generate RAS/CAS memory control signals and generate a WAIT signal that slows down the processor when it addresses slow EPROMs. Four TTL packages decode I/O addresses and provide Z-80 compatible control signals to the peripheral devices. Finally, one TTL package interfaces eight switches to the CPU without using up any of the more versatile PIO pins.

General Structure

Figure 1 shows the block diagram of this design. The Z8002 16-bit microprocessor is the heart of the system. This high-performance CPU offers a regular, easy to use architecture, an instruction set more powerful than that of many minicomputers, a sophisticated interrupt structure, and high throughput at a modest 4 MHz clock rate that allows a memory cycle time of 750 ns.

For a detailed description of the Z8002 that includes an instruction list with execution times for each instruction and each addressing mode, see the Z8001/Z8002 CPU Product Specification (03-8002-01). For a very detailed description of the Z8000 instruction set, refer to the Z8000 PLZ/ASM Assembly Language

Manual (03-3055-01).

Fixed program and data information is stored in an array of 2K x 8 ROMs or EPROMs; 16 16K x 1 dynamic RAMs provide 32K bytes of read/write storage. Input/output is handled by five I/O devices. Two Z-80A PIOs provide 4-byte-wide bidirectional ports (32 lines) with handshake control. A Z-80A SIO provides two fully independent full-duplex asynchronous or synchronous serial data communications channels. Four counter/timers in the Z-80A CTC relieve the processor from simple counting and timing tasks and generate the programmable baud-rates for the serial I/O channels. Eight switches can be interrogated and interpreted by the program.

Figure 1. Block Diagram

General Structure (Continued) The block diagram also indicates the various support functions. A crystal-controlled clock circuit generates a Z8002 and Z-80 compatible clock signal plus two complementary TTL clocks. Address buffers drive the memory and I/O devices; address latches demultiplex the time-shared Address/Data bus.

The ROM array uses a 1-of-8 address Decoder and the RAMs are driven by an address multiplexer and a RAS/CAS circuit. The timing for all these functions originates in the bus control and timing circuit. The I/O devices are selected by an I/O decoder and receive Z-80 equivalent control signals generated by the Z8002 to Z-80 Control Translator. The following sections contain detailed descriptions of these circuits.

Clock Generation

The Z8002 requires a continuously running clock with a frequency between 500 kHz and 4 MHz. Most Z8002 applications are performance oriented and the clock rate is therefore usually set close to the maximum limit of 4 MHz. At this frequency, the specified requirements for clock width (minimum of 105 ns High or Low) and clock transition times (maximum of 20 ns rise or fall) require careful attention. At 4 MHz, a 50% clock duty cycle is indirectly implied by this specification and the safest way to insure it is to start with a crystal oscillator frequency that is twice the clock

rate, and divide it with a toggling flip-flop. If this is impractical, a dc servo circuit can sense the average dc clock voltage level and adjust it to Vcc/2 by changing the oscillator bias.

The Z8002 clock input is not TTL compatible. It requires a High level within 400 mV of $V_{\rm CC}$. A resistive pull-up can achieve this level, but cannot guarantee the required rise-time (20 ns from 0.8 to 4.0 V) when driving the \approx 30 pF clock input capacitance. The stringent rise time requirements dictate the use of an active pull-up as shown in Figure 2.

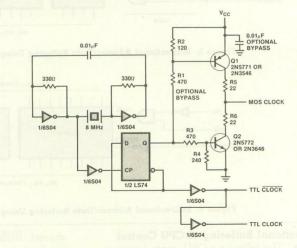


Figure 2. Clock Generation

CPU Output Buffering

The Z8002 outputs can sink 2 mA while maintaining TTL noise margins and can thus drive five LS-TTL inputs. All output delays are specified for a capacitive load of up to 100 pF. They increase by approximately 0.1 ns/pF of additional capacitive load.

Very small systems can be built without TTL buffering of the CPU outputs, but most systems require TTL buffering of the Address/Data lines and major control outputs, like AS DS MREO and R/W

Bidirectional buffering of the A/D lines. The Address/Data lines require Bus Transceivers, such as the LS243 Quad Non-Inverting Bus Transceiver with separate Enable inputs for the two directions (one active High: the other active Low), or the LS245 Octal Non-Inverting Bus Transceiver with a Direction Control input and an active-Low Enable input.

Figure 3 shows the logic that controls four LS243 Quad Transceivers; Figure 4 shows the

even simpler logic that controls two LS245

Octal Tranceivers.

The bus transceivers are controlled by three CPU control outputs as shown in the following truth table

| BUSAK | R/W | DS | |
|-------------|-------------|--|--|
| Н | Н | L Enable Receiver (input Data into CPU) | |
| H H H | H L L | H Enable Transmitter H (output Address or Data L from CPU) | |
| L | X | X Disable Transceiver | |

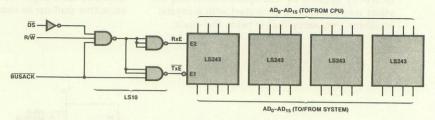


Figure 3. Bidirectional Address/Data Buffering Using Quad Transceivers

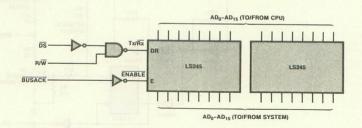


Figure 4. Bidirectional Address/Data Buffering Using Octal Transceivers

Unidirectional Buffering of CPU Control Outputs. The following CPU control outputs may require unidirectional buffering: AS, DS, MREQ, R/W, N/S, B/W.

The buffered signals must be 3-stated when BUSACK is Low. One LS365A or LS367A Hex 3-State Buffer can perform this function as shown in Figure 5. The LS244 Octal 3-State Buffer buffers eight signals, but uses a 20-pin

In a simple system, such as the one described here, BUSREQ is not used, so BUSACK is therefore always High. In a more complex system with direct memory access, a Low on BUSACK indicates that the CPU has relinquished the bus. If the buffered bus is

shared, BUSACK must be used to control the transceivers, as shown in Figures 4 and 5.

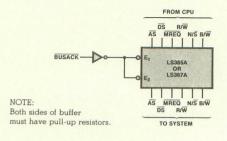


Figure 5. Control Signal Buffering

Address
Latching
(Demultiplexing the
A/D lines)

The Z8002 uses a 16-bit time-shared Address/Data bus that must be demultiplexed, that is, latched for use with standard (not edge-activated) memories. \overline{AS} is the obvious control signal for address latching and two LS373 Octal Transparent Latches are the best choice for this function (Figure 6). Note that

addresses are not guaranteed valid when \overline{AS} goes Low. It is therefore not possible to use the falling edge of \overline{AS} to clock the addresses into edge-triggered registers. The rising edge of \overline{AS} may be used as a clock, but this delays address availability by almost 100 ns. Transparent latches are the better choice.

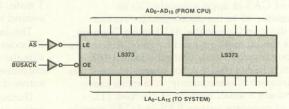


Figure 6. Address Latches

ROM Addressing

All microprocessors use nonvolatile memory for part of their program memory. Since the program status information after Reset is read from locations 0002 and 0004, it is natural to use the lower half of the addressing space for ROM or EPROM.

This application uses 2716-type 2K x 8 EPROMs addressed by the latched addresses LA_1 - LA_{11} . Pairs of 2716s store the low and high byte of each word. A_0 is ignored since the Z8002 always reads a full word from memory. LA_{15} must be used as a Chip Select input to separate the ROM and RAM areas. When more than 2K words of ROM or EPROM are used, an LS138 one-of-eight decoder selects between the ROM and EPROM pairs.

When driven with a 4 MHz clock, the Z8002 requires a read access time (address valid from the CPU to data required into the CPU) of 400 ns. After subtracting a 20 ns propagation delay through the LS373 address latches and an 18 ns propagation delay through the LS243 transceivers, the ROM or EPROM must have an access time (address in to data out) of better than 362 ns. Most ROMs and EPROMs have a longer access time and therefore require an additional wait state that relaxes the access time requirement by an additional 250 ns. Figures 8 and 9 show a 2-input NAND gate that generates a Wait signal whenever LA15 is Low and Q2 is High, thereby adding a wait state to every ROM/EPROM access.

RAM Address Multiplexing and RAS CAS Generation Dynamic 16K x 1 RAMs such as the Z6116 provide the most efficient read/write random-access store. Sixteen of these devices populate the upper half of the addressable memory space (LA₁₅ = High). Dynamic 16K RAMs use address multiplexing to reduce the package pin count, thus requiring only seven address inputs plus strobe inputs \overline{RAS} and \overline{CAS} .

Address Multiplexing. Two LS157 Quad Two-Input Multiplexers route the 14 address outputs $\underline{LA_1}$ - LA_{14} into the seven RAM address inputs. \overline{MREQ} synchronized with the rising clock edge is a convenient signal to control this multiplexer (Figure 7). LA_0 controls \overline{CAS} as described later.

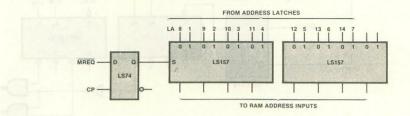


Figure 7. Address Multiplexer

RAM Address Multiplexing (Continued) RĀS and CĀS Generation. The most demanding signals in a dynamic memory design are the address strobes RĀS and CĀS, which must be timed carefully with respect to the address information and the multiplexer control.

Conceptually, MREQ might be used as RAS and DS as CAS. This would, however, require a memory read access time from the falling edge of CAS of approximately 120 ns (parameter 33 in the Z8001/Z8002 Product Specification Composite AC Timing Diagram, minus the 30 to 40 ns used by the CAS drivers and bus transceivers). Only the fastest 16K dynamic RAMs (6116-2, for example) meet this requirement. Consequently, it is more practical to use a small amount of clocked TTI. logic to generate earlier RAS and CAS signals and thus relieve the access time requirements so that even slow 16K RAMs (6116-3, -4 and -5) can be used. Figure 8 shows the circuit that generates RAS and CAS.

RAS is a 2-clock period (500 ns at 4 MHz) wide active-Low signal starting on the falling clock edge when AS is Low. The address information is valid and stable during the specified hold time (<50 ns) immediately after the falling edge of RAS. RAS is generated by an LS109 edge-triggered dual JK flip-flop, clocked by CP (that is, of a polarity opposite to the Z8002 clock). At the end of a machine cycle both Q1 and Q2 are High. The falling

edge of Clock during \overline{AS} clocks Q1 Low. The next falling clock edge leaves Q1 unaffected, but clocks Q2 Low. The next falling edge clocks Q1 High and leaves Q2 unaffected. The next falling clock edge clocks Q2 High and leaves Q1 High unless \overline{AS} is Low, in which case the cycle is repeated. Q1 is Low from the center of the first to the center of the third T state. Q2 is Low from the center of the second to the center of the fourth T state.

The left half of the LS139 Dual One-of-Four Decoder generates \overline{CAS} by ANDing three signals: LA₁₅; MREQ, which has been synchronized with Clock; and an auxilliary signal active during Read or \overline{DS} .

During a read operation, $\overline{\text{CAS}}$ becomes active at the beginning of $\overline{\text{T}_{2}}$; that is, on the rising edge of Clock after $\overline{\text{MREQ}}$ has gone Low. During a write operation, $\overline{\text{CAS}}$ is delayed until the beginning of $\overline{\text{DS}}$, when output data is guaranteed valid. The flip-flop is optional and stretches the width of $\overline{\text{DS}}$, thus stretching $\overline{\text{CAS}}$ (during write operations) from 160 to 200 ns, as required by the slower memories.

The right half of the LS139 Decoder controls the routing of \overline{CAS} to the two memory byte banks. The Z8002 addresses memory as bytes, but usually accesses words, ignoring A_0 . It uses A_0 only when writing a byte, in which case it suppresses \overline{CAS} to the byte bank that is not being written.

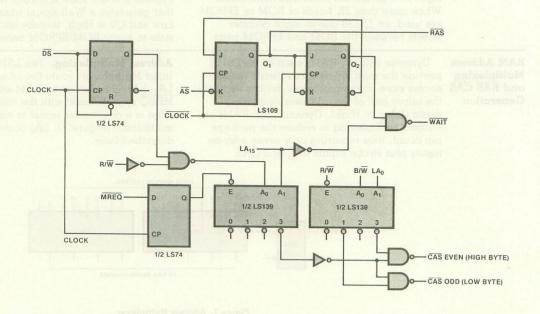


Figure 8. RAS, CAS, and WAIT STATE Generators

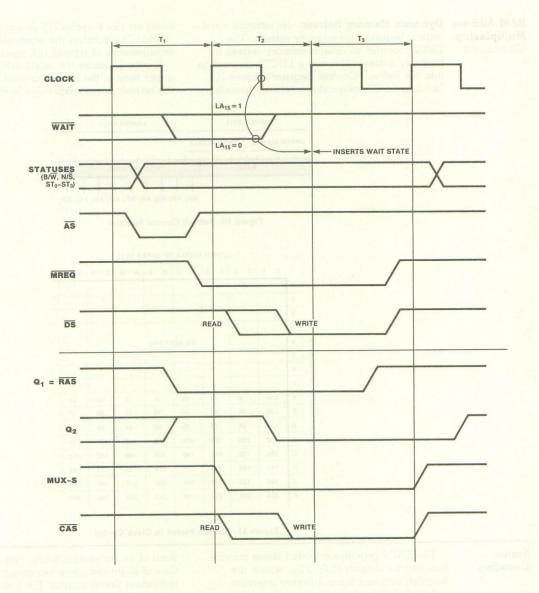


Figure 9. RAS and CAS Generation

Multiplexing (Continued)

RAM Address Dynamic Memory Refresh. No external hardware is required for memory refresh. The Z8002 provides automatic memory refresh if properly initiated through a LDCTL instruction into the Refresh Control Register (Figure 10). Loading a 9E00 generates a refresh operation

every 60 clock cycles (15 µs with a 4 MHz clock). This satisfies the worst-case refresh requirements of typical 16K dynamic RAMs.

Figure 11 shows the relationship between the upper byte of the refresh control register and the refresh period expressed in clock cycles.

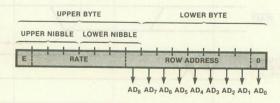


Figure 10. Refresh Control Register

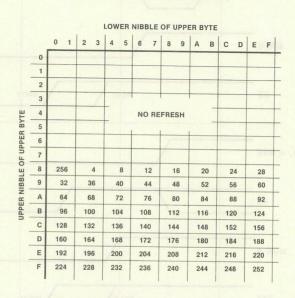


Figure 11. Refresh Period in Clock Cycles

Status Decoding

The Z8002 provides encoded status information on four outputs (ST₀-ST₃), which distinguish between three different interrupt acknowledge cycles; memory refresh; I/O reference; internal operation; data memory, stack or program memory access; and the first

word of an instruction fetch. Two LS138 One-of-Eight decoders can generate all the individual Status signals. For a simple system, only the first ten status codes have to be decoded. A single LS42 One-of-Ten Decoder is sufficient for this purpose (Figure 12).

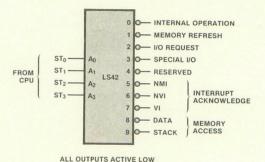


Figure 12. Status Decoder

Interfacing Peripheral Devices Z-Bus compatible peripheral devices that require no external logic to interface with a Z8002 will become available in the near future. In the meantime, this application note describes the use of Z-80 peripherals (Z-80A PIO, Z-80A CTC, Z-80A SIO) with the Z8002. These peripherals require only three additional TTL packages and a few lines of code to make the Z8002 emulate the typical Z-80 control signals \overline{IORQ} , $\overline{M1}$, \overline{RD} , and \overline{RETI} .

Four different operations are performed between the CPU and its peripherals:

- □ CPU writing into the peripheral device
- ☐ CPU reading from the peripheral device
- Peripheral interrupting the CPU, which responds with an Interrupt Acknowledge.
- □ CPU issuing a Return from Interrupt (RETI) signal.

The first two operations—writing to or reading from the peripheral—are fairly straightforward. An LS138 One-of-Eight Decoder, enabled by the decoded Status signal \overline{IORQ} , decodes the latched I/O address and generates \overline{CE} signals to the individual peripheral devices (Figure 13). The Z8002 can use the full 16-bit address space for I/O, but this application uses only LA₃-LA₁₀. When necessary, the higher-order address bits can also be decoded and fed into one of the Enable inputs.

A write operation into the enabled peripheral is performed when \overline{IORQ} is Low while \overline{RD} is High. Similarly, a read operation from the enabled peripheral is performed when \overline{IORQ} is Low while \overline{RD} is Low. As described later and shown in Figure 13 \overline{IORQ} and \overline{RD} are controlled by an LS352 Dual Four-Input Multiplexer enabled by a Strobe signal that is active from the beginning of T_2 to the end of T_3 , that is, for three clock periods (T_2 , T_{WA} , and T_3).

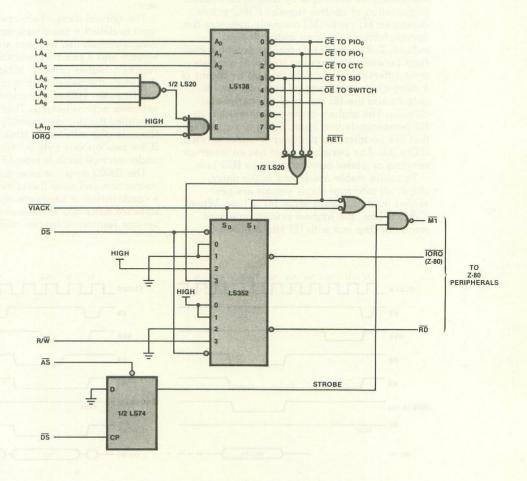


Figure 13. Interfacing with Z-80 Peripherals

Interfacing Peripheral Devices (Continued) The first four I/O devices addressed when LA₅ is Low are the four Z-80 devices (PIO 0, PIO 1, CTC, and SIO). The fifth peripheral is a set of eight switches that can be read by the CPU, which addresses them as peripheral. The user can thus specify any one of 256 different conditions (for example, chosing between 16 different baud rates for each of the two serial I/O channels). The sixth $\overline{\text{CE}}$ output addresses a phantom peripheral called RETI, which is activated at the end of an interrupt service.

The interrupt operation requires some extra logic and software to make the Z-80 peripherals compatible with the Z8002. Z-80 peripherals request a vectored interrupt by pulling the VI input of the CPU Low. The CPU (7-80 or 78002) samples this input at a specified moment prior to the end of any instruction execution. The 78002 then acknowledges the interrupt with a specific Status code (VIACK). The Z-80, which has no dedicated Interrupt Acknowledge output, acknowledges interrupts by issuing an unusual combination of control signals: IORO active during an MI cycle (MI normally indicates the opcode fetch cycle of an instruction execution). Z-80 peripherals resolve potential conflicts between overlapping interrupt requests from different interrupting devices by means of a daisy-chain arrangement between the IEO outputs and the IEI inputs of the peripheral devices. The highest-order peripheral has its IEI permanently tied High. For any peripheral that has no interrupt pending or under service. IEO = IEI. Any peripheral that has an interrupt pending or under service forces its IEO Low.

To insure stable conditions in the daisy chain, all interrupt status signals are prevented from changing while $\overline{\text{M1}}$ is Low. When $\overline{\text{IORQ}}$ is Low, the highest priority interrupt requestor (the one with IEI High) places its

interrupt vector on the data bus and sets its internal interrupt-under-service latch (Figure 14).

The circuit shown in Figure 13 generates the $\overline{\text{M1}}$, $\overline{\text{IORQ}}$, and $\overline{\text{RD}}$ signals required by the Z-80 peripherals during an interrupt acknowledge cycle. The timing for $\overline{\text{M1}}$ is generated by $\overline{\text{STROBE}}$, which goes High at the beginning of $\overline{\text{AS}}$ and stays High until the rising edge of $\overline{\text{DS}}$. An LS352 Dual 4-Input Multiplexer causes $\overline{\text{IORQ}}$ to go Low when $\overline{\text{DS}}$ is Low, and causes $\overline{\text{RD}}$ to remain High during the entire interrupt acknowledge cycle.

Return From Interrupt. At the end of an interrupt service routine the interrupt-underservice latch in the Z-80 peripheral that has been serviced must be reset. The Z-80 CPU accomplishes this by executing a special 2-byte instruction with the opcode sequence ED-4D (RETI) appearing on the data bus. All peripherals monitor this sequence and manipulate the daisy-chain in the following way:

The normal daisy-chain operation can be used to detect a pending interrupt; however, it cannot distinguish between an interrupt under service and a pending unacknowledged interrupt of a higher priority. Whenever "ED" is decoded, the daisy chain is modified by forcing High the IEO of any interrupt that has not yet been acknowledged. Thus the daisy chain identifies the device presently under service as the only one with an IEI High and an IEO Low. If the new opcode byte is "4D," the interrupt-under-service latch is reset (Figure 15).

The Z8002 does not have the equivalent RETI instruction and must therefore simulate it with a combination of hardware and software. A software sequence at the end of every interrupt service routine writes two consecutive bytes

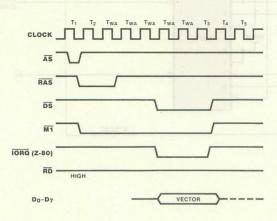


Figure 14. Interrupt Acknowledge Cycle

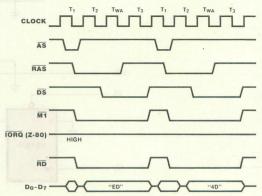


Figure 15. Return from Interrupt Cycle

Interfacing Peripheral Devices (Continued) (ED followed by 4D) into the phantom peripheral called RETI. The recommended software sequence is as follows:

| | , | |
|------|------------|----------------------|
| DI | | Disable Interrupts |
| LBD | RL1, #%ED | Load First Byte |
| OUTB | RETI, RL1 | Output First Byte |
| LDB | RL1, #%4D | Load Second Byte |
| OUTB | RETI, RL 1 | Output Second Byte |
| EI | | Enable Interrupts |
| RET | | Return From Interrup |
| | | |

During the two OUTB operations, each four clock cycles long, \overline{RETI} is Low, \overline{VIACK} is High and the following Z-80 control signals are generated: $\overline{MI} = \overline{STROBE}$, $\overline{IORQ} = \overline{High}$ and $\overline{RD} = \overline{DS}$. As described before, \overline{STROBE} is active during T_2 , T_{WA} , and T_3 , (for 750 ns at a 4 MHz clock rate).

Driving Z-80 Peripherals. The Z-80A PIOs, Z-80A CTC and Z-80A SIO are directly connected to the appropriate lines, as follows. The bidirectional AD_0 - AD_7 buffers are connected to the D_0 - D_7 data inputs/outputs on the peripherals.

The least significant address \underline{b} its LA_1 and LA_2 are used as Port Select (A/\overline{B}) and Control Data select (C/\overline{D}) on the PIO and SIO, and as Channel Select (CS_0 , CS_1) on the CTC.

The Interrupt outputs of all peripherals are interconnected (pulled up with a $4.7k\Omega$ resistor to $V_{\rm CC}$ and connected to the $\overline{\rm VI}$ input of the Z8002). The IEI-IEO interrupt daisy chain of the Z-80 peripheral devices must be connected appropriately to establish the desired hierarchy of interrupt priorities.

Conclusion

This Application Note demonstrates that a small, but powerful computer can be built around the Z8002 16-bit microprocessor using a very few standard TTL support packages. It

also shows how the readily-available Z-80 peripheral circuits interface easily to the Z8002, taking advantage of the similarity in the Z-80 and Z8000 interrupt structures.

| TTL Support Devices | Part Number | Function | Use | Figure |
|------------------------|-------------|---------------------|-------------------------------------|------------|
| | S04 | Hex Inverter | Clock Oscillator and Driver | 2 |
| | LS04 | Hex Inverter | Miscellaneous | 4, 5, 6, 8 |
| | LS74 | Dual Flip-Flop | Clock Divider and MREQ Synchronizer | 2 and 7 |
| | LS10 | Triple 3-Input NAND | Address/Data Buffer Control | 3 and 4 |
| | LS245 | Octal Buffer | Address/Data Buffer | 4 |
| | LS365A | Hex Buffer | Control buffer | 5 |
| | LS373 | Octal Latches | Address Latching | 6 |
| | LS373 | Octal Latches | Address Latching | 6 |
| | LS157 | Quad 2-Input MUX | Address Multiplexing | 7 |
| | LS138 | 1-of-8 Decoder | ROM Address Decoding | _ |
| | LS109 | Dual Flip-Flop | RAS Generation | 8 |
| | LS139 | Dual 1-of-4 Decoder | CAS generation | 8 |
| | LS74 | Dual Flip-Flop | DS Stretcher and STROBE Generator | 8 and 13 |
| | LS00 | Quad 2-Input NAND | CAS and WAIT gating | 8 |
| | LS42 | 1-of-10 Decoder | Status decoding | 12 |
| | LS138 | 1-of-8 Decoder | I/O decoding | 13 |
| | LS20 | Dual 4-Input NAND | I/O control | 13 |
| | LS352 | Dual 4-Input MUX | Control translator | 13 |
| | LS00 | Quad 2-Input NAND | Control translator | 13 |
| | LS244 | Octal Buffer | Switch buffer | |

Zilog Z8000 Family Technical Overview

CONTENTS

| Zilog Z8000 Family, The Art of Staying a Generation Ahead . | 615 |
|---|-----|
| Z8001/8002 CPU, Central Processing Unit | 617 |
| Z8010 MMU, Memory Management Unit | 619 |
| Z-bus System Structure | 621 |
| Z8034 UPC, Universal Peripheral Controller | 623 |
| Z8 MCU, Microcomputer Unit | 625 |
| Z8036 CIO, Counter / Timer and parallel I/O Unit | 627 |
| Z8030 SCC, Serial Communications Controller | 629 |
| Z6132 4K×8 Quasi-Static RAM | 631 |
| Z8038 FIO, FIFO Input/Output Interface Unit | 633 |
| Z8060 FIFO, FIFO Buffer Unit and Z-FIO Expander | 635 |
| More to Come | 636 |



The Art of Staying a Generation Ahead

If you know much about microprocessors, you know that Zilog became a technology leader by introducing the 8-bit Z-80 Family. The soundness of that family design has been amply proven, and its popularity is still growing.

Naturally, you expect Zilog to come up with more than just an extension of an 8-bit architecture to 16 bits. And you are right. The new Z8000 Family of microprocessor components is a whole generation ahead. Now you can design advanced concepts from the mainframe and minicomputer worlds into microcomputer systems that do more at less cost than ever before.

You won't find anything like this anywhere but Zilog. The advanced architecture, high throughput, intelligent peripherals and system flexibility of the Z8000 Family are sure to make it the most popular microprocessor of the 1980s. Want to find out why? Get acquainted with the family. Read on.

System Flexibility. The Z-Family spans the gulf between simple stand-alone computers and complex multiple-processor systems. Your system can grow as your application matures or expands.

Even the smallest Z8000 systems offer high throughput and easy programming far superior to any existing microprocessor alternative. In mid-range applications, Z8000 components offer much more powerful solutions to the design problems of word processing, intelligent terminals, data communications, instrumentation and process control. And in a complex network of multiple processors, smart peripheral components, small local memories and a large common memory, the Z8000 Family provides performance and versatility exceeding that of much larger-and far more expensive-minicomputers.

Higher Throughput, Reduced Cost. The powerful instruction set, high execution speed, regular architecture and numerous special features of the Z8000 CPU dramatically increase system throughput. Intelligent Z8000 peripheral controllers unburden the CPU and boost throughput even more.

Simply put, the Z8000 Family offers more for less money. The Z8000 CPU gives mid-range minicomputer performance at microprocessor cost. At component prices, Z8000 peripheral controllers perform complex system functions that previously required an entire PC board. Memory costs are reduced by Z-Bus memories that require no external logic, and by the compact code and moderate clock rate of the Z8000 CPU.

The Z8000 Family is designed for multiple-processor operation—an economical way of greatly increasing system performance. Many special features for multiple Z8000 CPUs facilitate the design of multiple-processor systems that share access to a common memory. The Memory Management Unit can dynamically relocate code and protect memory areas. The Z-UPC Universal Peripheral Controller, a complete slave microcomputer, can manipulate data off-line. Asynchronous parts of multipleprocessor systems can be joined by the Z-FIO FIFO Interface Unit. 7-Bus compatible memories make small local memory for dedicated CPUs an affordable item.

An Unmatched CPII. The 78000 CPU is far more than a wider data path, more registers, more data types, more addressing modes. more instructions and more addressing space. It brings the big-machine concepts to the level of microprocessors. The instruction set of the Z8000 CPU is more powerful than that of many minicomputers and incorporates features previously found only in large mainframes. Its generalregister architecture avoids bottlenecks associated with dedicated or implied registers. Special features support parallel processors, operating systems and compilers. For example, its ability to operate in both the System and Normal modes separates user programs from the operating system for better software security and modularity.

The Z8000 CPU is also a very fast machine. Its throughput is greater than that of any other 16-bit microprocessor and many minicomputers. And the Z8000 CPU achieves this at a moderate 4 MHz clock rate that allows you the choice of slow-access, low-cost memories.

How to Manage Your Memory Better. Modern trends are toward systems that have large and growing memories, multiple users, complex programs and requirements for effective system security. These design problems pose questions not sufficiently answered by other microprocessor families.

Another example of the Z-Family commitment to advanced architectural concepts, the MMU provides flexibility in code segment relocation and sophistication in memory protection found nowhere else in the microprocessor world. This unique device encourages modular software development—a necessary trend as programs reach new levels of complexity.

You are free from specifying where information is actually located in the physical memory because the MMU makes software addresses totally independent from the actual physical memory address. Some existing microprocessor CPUs do have internal CPU relocation registers, but they are dedicated and support few segments, and these CPUs restrict memory protection. Not true for the MMU. Various MMU configurations can randomly relocate all 128 segments output by the Z8000 CPU in all its six addressing spaces and with various translation tables for each space.

But the MMU is far more than a relocation device. It offers you a host of memory protection features that allow the system to protect its software from unwanted uses and users. Segments can be specified as read-only to protect them from being overwritten, as system-only to protect the operating system from inadvertent user access, as execute-only, and so on. A write warning zone is especially useful in stack operations so the operating system can deal with growing stacks.

Peripheral Problem Solvers. Z8000 peripheral components are not dumb I/O circuits. They perform intelligent, complicated tasks on their own. They unburden the CPU, reduce bus traffic and increase system throughput. Complex system tasks that previously required burdensome conglomorations of MSI, can now be handled off-line by Z-Bus peripherals with little CPU overhead.

Multifunction Z-Bus peripherals are extensively programmable, so each can be precisely tailored to its application. All share common interrupt and bus-request structures, as well as an I/O command structure that addresses up to 64 internal registers. Z-Bus peripherals can be operated in priority-interrupt or polled environments. They offer multiple channels to minimize chip count.

Counting, timing and parallel I/O problems seem less tiresome with the Z8036 Counter and Parallel I/O device. It has three 16-bit counter/timers, three I/O ports and can even double as a programmable interrupt-priority controller. Data communications are neatly handled by the Z8030 Serial Communication Controller. a dual-channel multi-protocol component that supports all popular communications formats Direct memory access is amply supported by the Z8016 DMA Controller, a fast dual-channel device that enhances the addressing power of the Z8000 CPU in stand-alone or parallel-processor environments. You can interface a variety of CRT displays with the Z8052 CRT Controller, which offers a vertical or horizontal split screen, oversize alphanumerics, smooth scrolling, and numerous other features. General-purpose control and datamanipulation problems are smoothly solved by the Z8034 Universal Peripheral Controller, a complete off-line microcomputeron-a-chip with three I/O ports. This processor executes the same friendly, capable instruction set as our Z8 Microcomputer. Bits and pieces of asynchronous parallelprocessing systems are interconnected by the Z8038 FIO Interface Unit, a surprisingly flexible device that can interface any major microprocessor and most peripherals to the Z-Bus. Its buffer depth can be expanded without limit using the Z8060 FIFO. Small local memory is affordable because the Z6132 4K x 8 Quasi-Static Z-Bus RAM requires no external interface circuitry, and it refreshes itealf



Features

- Regular, easy-to-use architecture.
- Instruction set more powerful than many minicomputers.
- Directly addresses 8M bytes.
- Eight user-selectable addressing modes.
- Seven data types that range from bits to 32-bit long words and word strings.
- System and normal operating modes; separate code, data and stack spaces.

- Sophisticated interrupt structure.
- Resource-sharing capabilities for multiprocessing systems.
- Multi-programming and compiler support.
- Memory management and protection provided by Z8010 Memory Management Unit.
- 32-bit operations, including signed multiply and divide.
- Z-Bus compatible.

Description

The Z8000 is an advanced high-end 16-bit microprocessor that spans a wide variety of applications ranging from simple stand-alone computers to complex parallel-processing systems. Essentially a monolithic minicomputer central processing unit, the Z8000 CPU is characterized by an instruction set more powerful than many minicomputers; resources abundant in registers, data types, addressing modes and addressing range; and a regular architecture that enhances throughput by avoiding critical bottlenecks such as implied or dedicated registers.

CPU resources include sixteen 16-bit general-purpose registers, seven data types that range from bits to 32-bit long words and word strings, and eight user-selectable addressing modes. The 110 distinct instruction types can be combined with the various data types and addressing modes to form a powerful set of 414 instructions. Moreover, the instruction set exhibits a high degree of regularity: most instructions can use any of the five main addressing modes and can operate on byte, word and long-word data types.

The CPU can operate in either system or normal modes. The distinction between these two modes permits privileged operations, thereby improving operating system organization and implementation. Multiprogramming is supported by the "atomic" Test and Set instruction; multiprocessing by a combination of instruction and hardware features; and compilers by multiple stacks, special instructions and addressing modes.

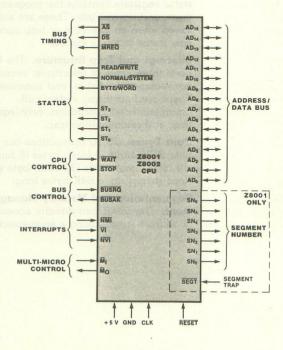


Figure 1. Pin Functions

The Z8000 CPU is offered in two versions: the Z8001 48-pin segmented CPU and the Z8002 40-pin non-segmented CPU. The main difference between the two is in addressing range. The Z8001 can directly address 8M bytes of memory; the Z8002 directly addresses 64K bytes. The two operating modes—system and normal—and the distinction between code, data and stack spaces within each mode allows memory extension up to 48M bytes for the Z8001 and 384K bytes for the Z8002.

To meet the requirements of complex, memory-intensive applications, a companion memory-management device is offered for the Z8001. The Z8010 Memory Management Unit manages the large address space by providing features such as segment relocation and memory protection. The Z8001 can be used with or without the Z8010. If used by itself, the Z8001 still provides an 8M byte direct addressing range, extendable to 48M bytes.

Register Organization. The Z8000 CPU is a register-oriented machine that has sixteen 16-bit general-purpose registers. The Z8002 CPU has one stack pointer register, and the Z8001 has two.

Stacks. The Z8001 and Z8002 can use stacks located anywhere in memory. Two implied stack pointers are available: the system stack pointer and the normal stack pointer.

Refresh. The Z8000 CPU contains a counter that can be used to refresh dynamic memory automatically.

Program Status Registers. This group of status registers contains the program counter, flags, and control bits. These are automatically saved when an interrupt or trap occurs, and a new status group is loaded.

Interrupt and Trap Structure. The CPU supports three types of interrupts: vectored and nonvectored maskable, and nonmaskable. There are four traps: system call, unimplemented instruction, privileged instruction, and segmentation trap.

Data Types. Z8000 instructions can operate on bits, BCD digits (4 bits), bytes (8 bits), words (16 bits), long words (32 bits), byte strings and word strings up to 64K bytes long.

Segmentation and Memory Management. The Z8001 can directly access 8M bytes of address space, using a segmented address-

ing scheme, implemented via the Z8010 MMU Memory Management Unit. The 8M bytes of Z8001 address space is divided into 128 relocatable segments of up to 64K bytes each. The addresses entered into instructions and output by the CPU in executing them are logical addresses. The MMU translates these logical addresses into addresses in physical memory. This process—relocation—is transparent to the user software.

Addressing Modes. Eight addressing modes are provided in the instruction set: Register (R), Immediate (IM), Indirect Register (IR), Direct Address (DA), Indexed (X), Relative Address (RA), Base Address (BA), and Base Indexed (BX)

Input/Output. A set of I/O instructions performs 8-bit or 16-bit transfers between CPU and I/O devices

Multimicroprocessor Support. A pair of CPU pins is used in conjunction with certain instructions to coordinate multiple microprocessors.

Instruction Set. The Z8000 has in its repertoire the nine categories of instructions following:

- Load and exchange
- Arithmetic
- Logic
- Program control
- Bit manipulation
- Rotate and shift
- Block transfer and string manipulation
- Input/output
- CPU control

Status Lines. Seven pins of the Z8000 are dedicated to the issuance of status information. Three are the function select lines Read/Write, Normal/System, and Byte/Word. The other four lines (ST_0 – ST_3) issue codes denoting type of operation (program or I/O reference, data or stack memory request, or internal operation), acknowledging external requests (segment trap or interrupt), and initiating memory refresh cycles.



Preliminary

Features

- Dynamic segment relocation makes software addresses independent of physical memory addresses.
- Sophisticated access validation protects memory areas from unauthorized or unintentional access.
- MMU architecture supports multiprogramming systems.
- Sixty-four variable-sized segments from 256 to 64K bytes can be managed within a total physical address space of 16M bytes; all 64 segments are randomly accessible.
- Multiple MMUs can support several translation tables for each of the six Z8001 address spaces.

Description

Declining memory costs coupled with the increasing power of microprocessors has accelerated the use of high-level languages, sophisticated operating systems, complex programs and large data bases in micromputer systems. The Z8001 microprocessor CPU supports these trends with an eight megabyte direct address space as well as a rich and powerful instruction set. The Z8010 Memory Management Unit (MMU) provides flexible and

efficient support for this large address space by offering dynamic segment relocation as well as numerous memory-protection features.

The primary memory of a computer is one of its major resources. As such, the management of this resource becomes a major concern as demands on it increase. These demands arise from multiple users (or multiple tasks within a dedicated application), the need to increase system integrity by limiting access

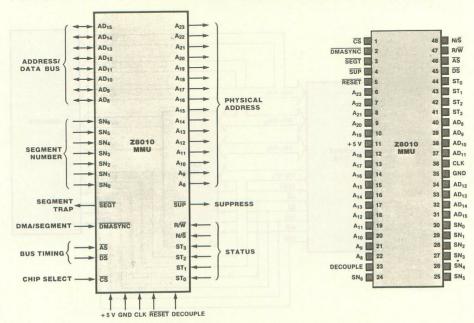


Figure 1. Pin Functions

Figure 2. Pin Assignments

to various portions of the memory, and from the need to structure large, complex programs and systems

Multiple tasks (or users) of a system that can reside anywhere in memory are called relocatable. Generally, systems in which all tasks are relocatable offer far greater flexibility in responding to changing system environments. Another aspect of multiple-task environments is sharing: separate tasks can execute the same program on different data, or several tasks may execute different programs using the same data.

Unfortunately, a problem that arises in multiple-task systems is that of system integrity. Tasks must be protected from unwanted interactions with other tasks; user tasks must be prohibited from performing operating system functions; and user tasks must also be protected from themselves so they cannot overflow the areas allotted to them.

In addition to these considerations, support for the design and implementation of large, complex programs and systems is itself an important consideration. Modern trends are toward the partitioning of a complex task into small, simple, self-contained subtasks that have well-defined interfaces. Because these subtasks interact with each other, communication between them must be carefully controlled. Memory-management systems can offer effective solutions for implementing large systems modularly designed.

The Z8010 Memory Management Unit supports multiple-process and large modular software systems with dynamic segment relocation. Futhermore, it enhances system integrity with

a powerful set of memory protection features.

Relocation. Dynamic segment relocation makes user software addresses independent of the physical memory addresses, thereby freeing the user from specifying where information is actually located in the physical memory and providing a flexible, efficient method for supporting multi-programming systems.

The Z-MMU uses a translation table to transform the 23-bit logical addresses from the Z8001 CPU into 24-bit addresses for the physical memory. Memory segments are variable in size from 256 bytes to 64K, in increments of 256 bytes. Pairs of Z-MMUs support the 128 segment numbers available for the various Z8001 CPU address spaces. Within an address space, any number of Z-MMUs can be used to accommodate multiple translation tables for system and normal operating modes, or to support more sophisticated memorymanagement systems.

System Integrity. Z-MMU memory-protection features safeguard memory areas from unauthorized or unintended access by associating special access restrictions with each segment. A segment is assigned a "personality" consisting of several attributes when it is initially entered into the Z-MMU. When a memory reference is made, these attributes are checked against the status information supplied by the Z8001 CPU. If a mismatch occurs, a trap is generated and the CPU is interrupted. The CPU can then check the status registers of the MMU to determine the cause and take appropriate action to correct the problem.

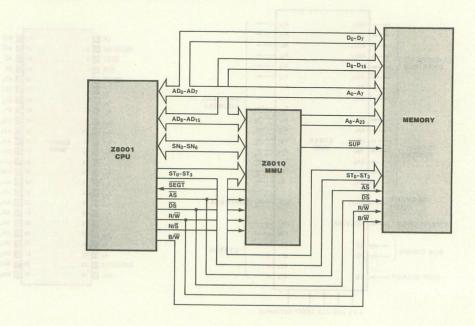


Figure 3. The MMU in a Z8000 System



Descriptive Brief

Preliminary

Features

- Multiplexed address/data bus, shared by I/O and memory.
- Peripherals may be asynchronous.
- Up to 24-bit memory address, 16-bit I/O.
- 8 or 16 data bits.

- Daisy-chained bus request.
- Daisy-chained resource request.
- Vectored or nonvectored interrupts.
- Separate memory and I/O address space.

Description

The Z-bus is a shared bus that links the components of the Z8000 family. A bus user can be any device that can generate bus transactions. Five different types of transactions can be passed on the Z-bus to serve the basic needs of I/O and memory structures in a distributed-processing environment. The five types are:

- Memory access
- I/O transfer
- Interrupt
- Bus request
- Resource request

Direct addressing of the internal registers of peripherals is facilitated by the use of multiplexed address and data lines. (See Figure 1.) The Z-bus is asynchronous, so peripherals' clocks need not be synchronized with the CPU clock, which is therefore not transmitted on the bus directly. The signals (strobes, acknowledges, etc.) generated in the course of any transaction provide all necessary timing information.

Memory Access. Status signals issued by the CPU distinguish memory transactions from others and select the address space to be accessed. Slow memory devices may assert the $\overline{\text{WAIT}}$ signal to prolong the transaction. Extended addresses may be used with the segmented Z8001 CPU and the Z8010 MMU. Other status signals define direction (R/\overline{W}) , Normal/System (N/\overline{S}) , Byte/Word (B/\overline{W}) , and the various address spaces.

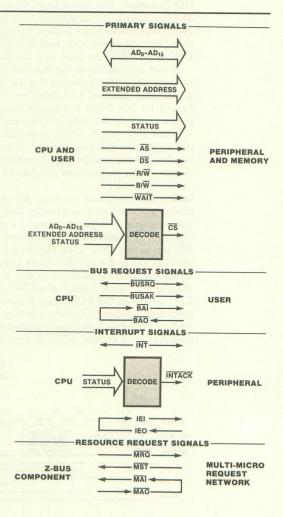


Figure 1. Z-Bus Signals

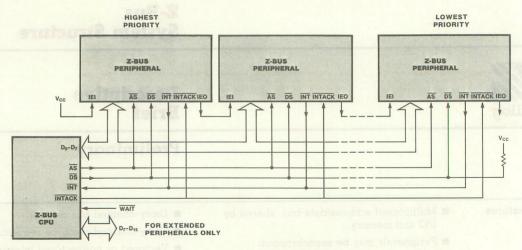


Figure 2. Interrupt Connections

I/O Transfer. The status line I/O reference distinguishes I/O transactions from others. The 16-bit multiplexed bus is used for address and data (without extension), and \overline{AS} , \overline{DS} , R/W, B/W, and \overline{WAIT} are used in a similar way.

Direct addressing of the internal registers of peripherals is facilitated by the use of multiplexed address and data lines. (See Figure 1.) The Z-bus is asynchronous, so peripherals' clocks need not be synchronized with the CPU clock, which is therefore not transmitted on the bus directly. The signals (strobes, acknowledges, etc.) generated in the course of any transaction provide all necessary timing information.

Interrupt. The Z-bus interrupt scheme is an interrupt-under-service priority daisy chain that requires no separate priority controller. Interrupt requests are all tied directly to the INT pin of the CPU. (See Figure 2.) Physical position along the IEI/IEO daisy chain determines the priority assigned to any given peripheral. Upon receipt of an INT signal, the CPU issues an INTACK. (See Figure 3.) This temporarily inhibits further interrupt requests, while all devices that have initiated interrupt requests prior to that INTACK drop their IEO

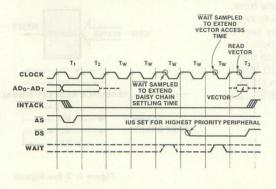


Figure 3. Interrupt Acknowledge Timing

outputs. (Multiple INTs might occur simultaneously.) The highest-priority IEO has the effect of removing IEI inputs from all devices beyond it on the same daisy chain, thereby preventing them from requesting interrupts further until their IEI inputs are restored. Three Wait cycles after the leading edge of INTACK (or more, if WAIT has been asserted by the highest-priority device requesting service), to allow the chain to settle, a DS from the CPU stimulates the one highest-priority requesting peripheral to place its vector on the bus. Two (or more) additional Wait cycles later, the service routine is invoked, and INTACK is returned high. At this time, all requesters of higher priority than the one being serviced (those whose IEI lines are still high) are enabled, and may generate new interrupt requests. Once a peripheral has been serviced it unmasks the daisy chain so lower-priority interrupts can be generated.

Bus Request. The bus request is used to transfer control of the Z-bus for memory or I/O transactions. The \overline{BUSRQ} input line to the Z-bus CPU, the wired-OR of \overline{BRQ} outputs from all requesters, initiates a bus request. The \overline{BUSAK} output line from the CPU is daisy-chained through \overline{BAI} inputs and \overline{BAO} outputs of all requesters in order of priority, to grant use of the bus to the first requester whose \overline{BAO} is held high at that time.

Resource Request. The resource request chain is used to share a resource among several Z-bus CPUs, none of which is default master of that resource. The resource-request protocol is similar to that of the bus request, except for an added status line that inhibits all requesters from issuing requests any time the resource is busy. The acknowledge daisy-chain resolves contention in the event of simultaneous requests.



Preliminary

Features

- Complete slave microcomputer, for distributed-processing Z-bus use.
- Unmatched power of Z8 architecture, instruction set.
- Three programmable I/O ports, two with 2-wire handshake, or any combination of data and control lines.
- Six levels of priority interrupts to Z-UPC.
- Two programmable 8-bit counter/timers with 6-bit prescalers.
- 256 byte register file, accessible by both master CPU and Z-UPC, as allocated by Z-UPC program.
- 2K bytes of on-chip program ROM for efficiency, versatility.

Description

The Z-UPC Universal Peripheral Controller is a distributed microcomputer that performs the three basic interfacing functions needed to interface a CPU with peripherals: device control by ROM-resident internal software, data manipulation, such as reformatting or arithmetic, and data buffering in internal registers.

The Z-UPC is similar to the Z8 microcomputer and uses the Z8 instruction set. Under

program control, its three 8-line I/O ports can be tailored to the needs of its user. Permanently configured as a single-chip controller with 2K bytes of internal ROM, the Z-UPC executes instructions in $2.2~\mu s$ average using a 4-MHz clock source. Its register file contains 256 bytes, of which 234 are general-purpose registers, 19 are status and control registers, and three are port registers.

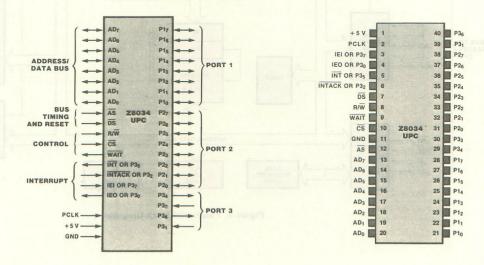


Figure 1. Pin Functions

Figure 2. Pin Assignments

The Z-UPC Universal Peripheral Controller is an intelligent device that generates all the control signals peripheral devices need. Because it does off-line arithmetic, translates data before transmitting, and buffers data, the Z-UPC unburdens the master CPU, thereby increasing the overall speed and efficiency of the system in which it resides.

Based upon the Z8 microcomputer architecture, the Z-UPC offers fast execution time, efficient use of memory, and sophisticated interrupt, I/O, and bit manipulation. Its powerful and extensive instruction types, combined with its efficient internal register addressing scheme, not only speeds program execution, but also efficiently packs program into the onchip ROM.

A unique characteristic of the Z-UPC is its register file, which contains I/O port and control registers that can be accessed both by the Z-UPC program and by its associated master CPU. This results in byte efficiency, programming efficiency, and address space efficiency because Z-UPC instructions can operate directly on I/O data without moving it to and from an accumulator. It also allows the Z-UPC user to allocate as data buffer between the CPU and

the peripheral all register space not in use as accumulators, address pointers, index registers, or stack. Registers not used as buffer are protected against CPU access. The register file is divided into 16 groups of 16 working registers each. A register pointer uses fast, shortformat instructions to access any one of these groups guickly, resulting in fast and easy task switching. Two-way communication between the master CPU and the register file is facilitated by another pointer that positions 16 interface registers anywhere within the register file. These registers are accessed directly by both the master CPU and the slave Z-UPC. Four more registers, similarly accessed, convev control and status information.

All of the Z-bus's daisy-chained priority interrupt system can be implemented in the Z-UPC under software control, or the Z-UPC can be programmed to function in a polled environment. In all, the Z-UPC has 24 pins that can be dedicated to I/O functions. Grouped logically into three 8-line ports, they can be programmed in many combinations of inputs, outputs, and bidirectional lines, with or without handshake and with push-pull or opendrain outputs.

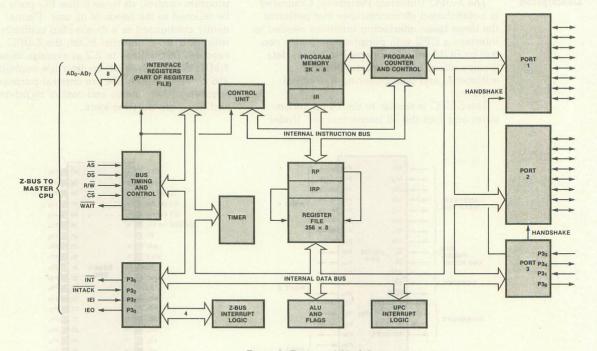


Figure 3. Functional Block Diagram



Preliminary

Features

- Complete microcomputer, 2K bytes of ROM, 32 I/O lines, and up to 62K addressable external space each for program and data memory.
- 144-byte register file, including 124 general-purpose registers, 4 I/O ports registers, and 16 status and control registers.
- Full-duplex UART and two programmable 8-bit counter-timers, each with a 6-bit programmable prescaler.
- Register pointer so short, fast instructions

- can access any of nine working-register groups in 1.5 $\mu \mathrm{s}.$
- Average instruction execution time of 2.2 μ s, maximum of 4.25 μ s.
- Vectored, priority interrupts for I/O, counter-timers, and UART.
- On-chip oscillator which accepts crystal, RC, or LC reference or external clock drive.
- Low-power standby option which retains contents of general-purpose registers.

Description

The Z8 microcomputer introduces a new level of sophistication to single-chip architecture. Compared to earlier single-chip microcomputers, the Z8 offers faster execution; more efficient use of memory; more sophisticated interrupt, input/output and bit-manipulation capabilities; and easier system expansion.

Under program control, the Z8 can be tailored to the needs of its user. It can be con-

figured as a stand-alone microcomputer with 2K of internal ROM, a traditional microprocessor that manages up to 124K of external memory, or a parallel-processing element in a system with other processors and peripheral controllers linked by the Z-Bus. In all configurations, a large number of pins remain available for I/O.

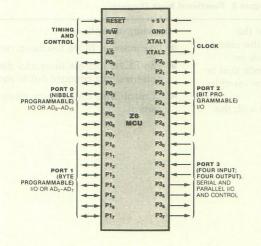




Figure 1. Pin Functions

Figure 2. Pin Assignments

Z8 architecture is characterized by a flexible I/O scheme, an efficient register and address space structure and a number of ancillary features that are helpful in many applications.

Microcomputer applications demand powerful I/O capabilities. The Z8 fulfills this with 32 pins dedicated to input and output. These lines are grouped into four ports of eight lines each and are configurable under software control to provide timing, status signals, serial or parallel I/O with or without handshake, and an address/data bus for interfacing external memory.

Because the multiplexed address/data bus is merged with the I/O-oriented ports, the Z8 can assume many different memory and I/O configurations. These configurations range from a self-contained microcomputer to a

microprocessor that can address 124K of external memory.

The Z8 offers three basic address spaces to support this wide range of configurations: program memory (internal and external), data memory (external) and the register file (internal). The 144-byte random-access register file is composed of 124 general-purpose registers, 4 I/O port registers, and 16 control and status registers.

To unburden the program from coping with real-time problems such as serial data communication and counting/timing, the Z8 offers an on-chip asynchronous receiver/transmitter (UART), and two counter/timers with a large number of user-selectable modes. Hardware support for the UART is minimized because one of the on-chip timers supplies the bit rate.

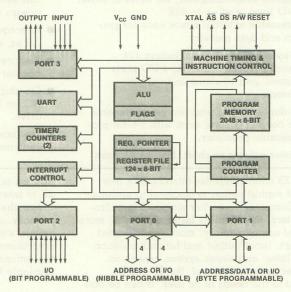


Figure 3. Functional Block Diagram

Z8/64 Development Device The 64-pin development version of the 40-pin mask-programmed Z8 allows the user to prototype the system in hardware with an actual Z8 device, and develop the code that is eventually mask-programmed into the on-chip ROM of the Z8/40.

The Z8/64 is identical to the Z8/40 with the

following exceptions:

- The internal ROM has been removed.
- The ROM address lines and data lines are buffered and brought out to external pins.
- Control lines for the new memory have been added.



Preliminary

Features

- Two independent 8-bit double-buffered bidirectional I/O ports plus a special-purpose 4-bit I/O port.
- Four handshake modes including IEEE-488.
- Wait/Request line for high speed data transfer.
- Three independent 16-bit counters.
- All registers read/write and directly addressable.
- Flexible pattern recognition logic, programmable as 16-input interrupt controller.

Description

The Z8036 CIO Counter/timer and Parallel I/O element is a general purpose peripheral circuit that satisfies most counter/timer and parallel I/O needs encountered in system designs. This versatile device contains three I/O ports and three counter/timers. Many programmable options tailor its configuration to specific applications. The use of the device is simplified by making all internal registers (command, status, and data) readable and (except for status bits) writable. Also, each register is given its own unique address so it can be accessed directly—no special sequential operations are required. The Z-CIO is directly Z-bus compatible.

Either 8-bit I/O port can be a handshake

byte port or a bit port. In the bit mode, data direction is programmable bit by bit. In the handshake mode, the ports can be input, output, or bidirectional, and they may be linked to form a 16-bit port. The four handshake modes include IEEE-488, interlocked (for interfacing to a Z-UPC, Z-FIO or another Z-CIO), strobed and pulsed. The pulsed mode connects one counter/timer with the handshake logic for interfacing a mechanical device such as a printer. The 4-bit port provides handshake controls, special controls (Wait/Request) or general-purpose I/O.

The counter/timer section contains three 16-bit counters, two of which can be software-configured as a 32-bit counter/timer. Up to

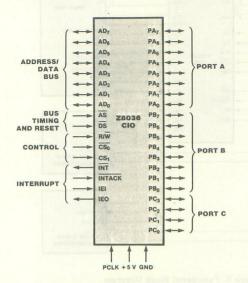


Figure 1. Pin Functions

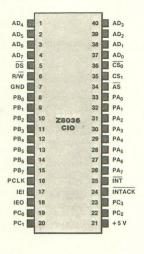


Figure 2. Pin Assignments

four I/O lines for each counter are available for direct external control and status information. All counters have a programmable output duty cycle, continuous or single-cycle operation, and the counting process can be programmed to be either retriggered or nonretriggered.

Figure 3 shows how the Z-CIO is used. The two general purpose 8-bit ports are similar. They can be programmed as handshake driven, double-buffered ports (input, output, or bidirectional) or as control ports in which the direction of each bit is individually programmable. Port B can also be specified to provide external access for two of the counter/timers. Each port includes pattern recognition logic allowing interrupt generation when a specified pattern is detected. The pattern recognition logic can be programmed so that the port functions like a priority interrupt controller

To control these capabilities, each port contains 13 registers. Three of these, the input. output, and buffer registers, are data path registers. Two others, the mode specification and handshake specification registers, define the mode of the port and specify what handshake to use, if any. The reference pattern for the pattern recognition logic is defined in three registers, the pattern polarity, pattern transition, and pattern mask registers. The detailed characteristics of each bit path (for example, the direction of data flow, or whether a path is inverting or noninverting) are programmed using the data path polarity, data direction, and special I/O control registers. The primary control and status bits are grouped in a single register so that after the ports are configured initially, only this register need be accessed often. One register contains the interrupt vector associated with each port. To facilitate initialization, the port logic is designed so that if a capability of the port is not required the registers associated with that capability are ignored and need not be programmed.

The function of port C depends upon the roles of ports A and B. Port C provides handshake lines for the other two when required. Any bits of port C not so used can be used as I/O lines or as external access to the third counter/timer.

Besides the data input and output registers, three registers are needed. These specify the details of each bit path: data path polarity, data direction, and special I/O control.

The three counter/timers are all identical. Each is composed of a 16-bit down-counter, a 16-bit time constant register (which holds the value loaded into the down-counter), a 16-bit current count register (used to read the contents of the down-counter), and two 8-bit registers for control and status (the mode select and control registers). All three share a common vector register.

Each counter/timer can be programmed as either counter or timer. Up to four port I/O lines can be designated as external access lines for it. The lines are: Counter Input, Gate Input, Trigger Input, and Counter/Timer Output. Three different counter/timer output duty cycles are available: pulse, one-shot, or square wave. The operation of the counter/timer can be specified to be either single cycle or continuous. The counting sequence may be retriggered or nonretriggered, under program control.

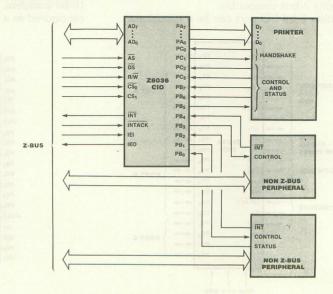


Figure 3. Functional Block Diagram



Preliminary

Features

- Two independent, 0 to 1 Megabit-persecond, full-duplex channels, each with its own quartz oscillator, baud-rate generator, and digital phase-locked loop for clock recovery.
- Multi-protocol operation under program control.
- Asynchronous mode with 5 to 8 bits and 1, 1½, or 2 stop bits per character; programmable clock factor; break detection and generation; parity, overrun, and framing error detection.
- Local loopback and auto-echo modes.

- Bisynchronous mode with internal or external character synchronization on one or two sync characters and CRC generation and checking with CRC-16 or CRC-CCITT preset to either 1s or 0s.
- SDLC/HDLC mode with comprehensive frame-level control, automatic zero insertion and deletion, I-field residue handling, abort generation and detection, CRC generation and checking, and loop mode operation.
- Programmable for NRZ, NRZI, or FM coding.

Description

The Z-SCC Serial Communication Controller is a dual-channel, multi-protocol data communication peripheral for Z-bus use. It is software-configured to satisfy a wide variety of serial communication applications. Its basic function is serial-to-parallel and parallel-to-serial conversion. However, the Z-SCC also contains a repertoire of new, sophisticated internal functions that minimize the need for

external random logic on the circuit card.

The Z-SCC handles asynchronous formats, synchronous byte-oriented protocols such as IBM Bisync, and synchronous bit-oriented protocols such as HDLC and IBM SDLC. This versatile device also supports virtually any other serial data transfer application (cassette or diskette interface, for example).

The device can generate and check CRC

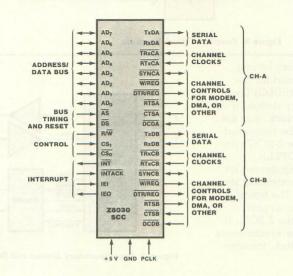


Figure 1. Pin Functions



Figure 2. Pin Assignments

codes in any synchronous mode and can be programmed to check data integrity in various modes. It also has facilities for modem controls in both channels. In applications where these controls are not needed, the modem controls can be used for general-purpose I/O.

As is standard among Zilog peripheral components, the Z-bus daisy-chain interrupt heirarchy is supported.

The Z-SCC contains the necessary multiplexed address/data bus interface with strobe and chip select lines to function as a Z-bus peripheral. It includes internal control and interrupt logic, two full-duplex channels and two baud-rate generators. Associated with each channel are several read and write registers for mode control as well as the logic necessary to interface to modems or other external devices.

The read and write register group for each channel includes eight control registers, two sync-character registers, and four status registers. Each baud rate generator has two read/write registers for holding the time constant that determines baud rate. Associated with the interrupt logic is a write register for interrupt vector and three read registers: vector with status, vector without status, and interrupt pending status.

The logic for both channels provides formatting, synchronization and validation for data transferred to and from the channel interface. The modem control inputs are monitored by the control logic under program control. All of the modem control signals are general purpose in nature and optionally can be used for functions other than modem control.

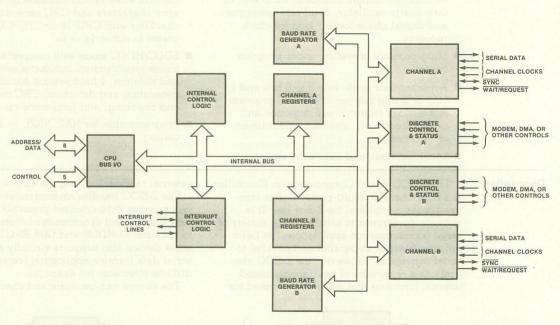


Figure 3. Functional Block Diagram

Typical Applications

Figure 4 shows how a Z-SCC can be connected with channel A programmed for the Synchronous Data Link Control (SDLC) Loop mode, functioning as a secondary station. If NRZI or FM coding is used, no clock lines are required because the clock can be recovered from the received data, using the Z-SCC's onchip digital phase locked loop (DPLL). Another Z-SCC (not shown), programmed for the SDLC mode, would be the controlling station, polling the loop for traffic. The figure shows a typical, asynchronous serial port being serviced by channel B of the Z-SCC. It could just as well support another synchronous data link, or even a high-speed link, transferring data via a DMA controller.

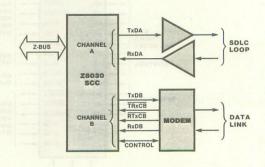


Figure 4. Loop Secondary Station and Serial Port



Preliminary

Features

- Byte-wide organization: 4096 words by eight bits.
- Access and cycle times guaranteed over voltage and temperature range:

| Part Number | Access Time | Cycle Time |
|-------------|-------------|------------|
| Z6132-3 | 200 ns | 325 ns |
| Z6132-4 | 250 ns | 375 ns |
| Z6132-5 | 300 ns | 425 ns |

■ Low power consumption: 200 mW active, 125 mW stand-by.

- Industry-standard 28-pin DIP with JEDEC-recommended pinout.
- ±10% tolerance on single +5 V supply voltage.
- Automatic self-refresh scheme with slow-and fast-cycle modes.
- All inputs and outputs are TTL compatible
- On-chip substrate bias generator.
- Interfaces readily to Z8 and Z8000.

Description

The Zilog Z6132 is a +5 V intelligent MOS dynamic RAM organized as 4096 words by eight bits. Although it uses single-transistor dynamic storage cells, the Z6132 effectively functions as a static RAM because it performs and controls its own refresh. This eliminates the need for external refresh circuitry and combines the convenience of a static RAM with the high density and low power consumption normally associated with a dynamic RAM.

The Z6132 is particularly suited for microprocessor and minicomputer applications where its byte-wide organization, transparent self-refresh operation and single supply voltage reduce the parts count and simplify the design.

The Z6132 uses high-performance depletionload double-poly n-channel silicon-gate MOS technology with a mixture of static and dynamic circuitry that provides a small memory cell, fast access and low power consumption. The Z6132 has separate pins for addresses and bidirectional data I/O to provide maximum flexibility in its application.

The circuit is packaged in an industrystandard 28-pin DIP and pin compatible with the proposed JEDEC standard.

The Z6132 conforms with the Z-Bus specification used by the new generation of Zilog microprocessors, the Z8 and Z8000.

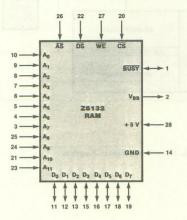


Figure 1. Pin Functions



Figure 2. Pin Assignments

The Z6132 4K x 8 quasi-static RAM is organized as two separate blocks, each having two sets of 64 rows on either side of the 128 sense amplifiers. Both blocks have separate and independent row address buffers and decoders, but they share the column decoder and the internal 8-bit wide data path. The two sets of row address decoders are addressed either by the address inputs A_1 - A_7 or by the internal 7-bit refresh counter. The least significant address input (A_0) selects one of the two blocks for external access. While the selected block performs a read or write operation, the other memory block uses the refresh counter address to refresh one row. Details of the self-refresh mechanism are explained later.

A memory cycle starts when the rising edge of Address Strobe (\overline{AS}) clocks in Chip Select (\overline{CS}), A_0 , and Write Enable (\overline{WE}). If the chip is not selected (\overline{CS} = High), all other inputs are ignored for the rest of the cycle (that is, until the next rising edge of \overline{AS}). Both memory blocks are self refreshed by the 7-bit refresh counter. If the chip is selected (\overline{CS} = Low), the 12 address bits and the Write Enable bit are clocked into their registers. A_0 determines which block is addressed by A_1 – A_{11} ; the other block is refreshed by the 7-bit refresh counter.

The Chip Select and Address inputs must be valid only during a short hold time after the rising edge of \overline{AS} . This allows address/data

multiplexing, because data I/O is controlled by a separate control input Data Strobe (\overline{DS}) .

Read Cycle. A read cycle is initiated by the rising edge of Address Strobe (\overline{AS}) while Chip Select (\overline{CS}) is Low and Write Enable (\overline{WE}) High. A Low level on the Data Strobe (\overline{DS}) input activates the Data outputs after a specified delay from the rising edge of \overline{AS} as well as the falling edge of \overline{DS} , whichever comes later. During a read operation, \overline{DS} is nothing but a static Output Enable signal.

Write Cycle. A write cycle is initiated by the rising edge of Address Strobe (\overline{AS}) while Chip Select (\overline{CS}) is Low and Write Enable (\overline{WE}) are Low.

The $\overline{\text{WE}}$ input is checked again at the beginning (falling edge) of Data Strobe ($\overline{\text{DS}}$). If $\overline{\text{WE}}$ is still Low, this falling edge of $\overline{\text{DS}}$ edgetriggers the data on the D_0-D_7 inputs into the addressed memory location. Data must be valid only during a short hold time after the falling edge of $\overline{\text{DS}}$.

Write Inhibit Cycle. After a write cycle has been initiated, the actual write operation can still be aborted by pulling \overline{WE} High again before the falling edge of \overline{DS} . This write inhibit cycle is a special feature that permits starting a write cycle early at \overline{AS} time, but still allows the option of inhibiting the write operation later at \overline{DS} time.

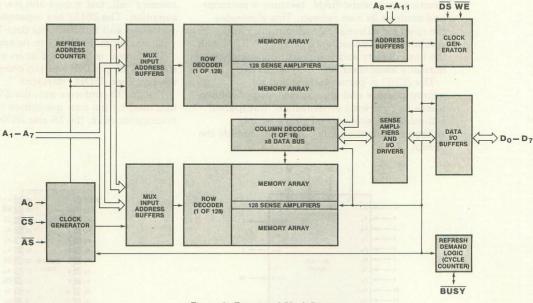


Figure 3. Functional Block Diagram



Preliminary

Features

- Asynchronous bidirectional FIFO buffer, used with most major microprocessors as CPU/CPU or CPU/peripheral interface.
- Interlocked or IEEE-488 handshake port mode; Empty, Full, and Wait/Request lines for high-speed data transfer.
- 128 x 8 organization, expandable to 16 bits wide; cascadable to any depth.
- Pattern recognition logic stops DMA transfer and/or interrupts CPU.
- Preset byte count in FIO buffer can interrupt CPU.
- All registers read/write and directly addressable.

Description

The Z-FIO is a general-purpose microprocessor interface that provides elastic buffering between asynchronous CPUs in a parallelprocessor network or between CPU and peripheral circuits. The Z-FIO can interface a Z-bus microprocessor or any other major processor to another microprocessor or to a peripheral circuit or port.

In Z8000 systems, the FIO furthers distributed-processor operation because it can interconnect components or subsystems operating at different speeds. Also, it can increase system throughput by transferring words as well as bytes. This bidirectional device accepts data and holds it until it can be used by another device in the system. In most I/O transactions, introducing a 128-deep buffer cuts interrupt servicing overhead by two orders of magnitude.

The Z-FIO greatly facilitates system throughput by moving variable-size blocks under either direct memory access or interrupt control—an especially important consideration when fast peripheral circuits need interfacing. Complete status information is also provided for operation in polled environments.

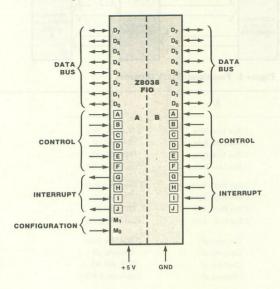


Figure 1. Pin Functions



Figure 2. Pin Assignments

| Pin Assignments | olaO\t | Z-Bus Low Byte | Z-Bus High Byte | General Purpose | Interlocked HS Port* | IEEE 488 HS Port* | |
|--------------------|--------|-------------------|--------------------|--------------------|-------------------------|----------------------|--|
| | A | REQ/WT | REQ/WT | REQ/WT | RFD/DAV | RFD/DAV | |
| | В | DMASTB | DMASTB | DACK | ACKIN | DAV/DAC | |
| | C | DS | DS | RD | FULL | DAC/RFD | |
| | D | R/W | R/W | WR | EMPTY | EMPTY | |
| | E | CS | CS | CS | CLEAR | CLEAR | |
| | F | ĀS | ĀS | C/D | DATA DIR | DATA DIR | |
| | G | INTACK | A ₀ | INTACK | IN ₀ | IN ₀ | |
| | H | IEO | Ai | IEO | OUT | OUT | |

IFI

INT

A₂

I

I

Description (Continued)

The internal functions of the Z-FIO are shown in the block diagram (Figure 3). It is made up of two sides that are identical except for programming. The side programmed by pins Mo and Mi is called the A side; the side programmed by bits SL₀ and SL₁ is called the B side. Common to both, and situated between the two sides, is the 128 × 8 RAM used for data storage, two 7-bit counters, and various registers. The RAM is capable of simultaneous, independent read and write operations. This means, for example, that the A side CPU can write a byte of data into the FIO without disturbing a simultaneous read operation by the B side CPU. The outputs of the read and write counters are used to address the buffer RAM,

IEO

INT

and also are fed into a subtractor to determine the current number of bytes in the memory. This number can be read by either CPU from a status register dedicated to each side. Another programmable register is compared against the status register to generate an interrupt and/or start and stop DMA transfers. A pair of port registers allows for communication between CPUs, bypassing the main buffer memory.

IN₂

OUT₃

INa

OUT₂

Operating Modes. The Z-FIO has twelve different programmable modes. (Table below.) The states of two package pins determine the mode of operation of the A side, and the B side is programmed by two bits (SL_0 and SL_1) in one of the A side control registers.

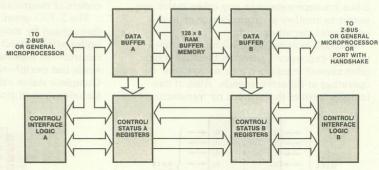


Figure 3. Functional Block Diagram

| Operating Modes - | Mode | M ₁ | M ₀ | SL_1 | SL ₀ | A | В | |
|----------------------|------|----------------|----------------|--------|-----------------|-----------------|-----------------|--|
| | 0 | 0 | 0 | 0 | 0 | Z-Bus Low Byte | Z-Bus Low Byte | |
| | 1 | 0 | 0 | 0 | 1 | Z-Bus Low Byte | General µP | |
| | 2 | 0 | 0 | 1 | 0 | Z-Bus Low Byte | IEEE 488 HS | |
| | 3 | 0 | 0 | 1 | 1 | Z-Bus Low Byte | Interlocked HS | |
| | 4 | 0 | 1 | 0 | 0 | Z-Bus High Byte | Z-Bus High Byte | |
| | 5 | 0 | 1 | 0 | 1 | Z-Bus High Byte | General μP | |
| | 6 | 0 | 1 | 1 | 0 | Z-Bus High Byte | IEEE 488 HS | |
| | 7 | 0 | 1 | 1 | 1 | Z-Bus High Byte | Interlocked HS | |
| | 8 | 1 | 0 | 0 | 0 | General μP | Z-Bus Low Byte | |
| | 9 | 1 | 0 | 0 | 1 | General µP | General μP | |
| | 10 | 1 | 0 | 1 | 0 | General μP | IEEE 488 HS | |
| | 11 | 1 | 0 | 1 | 1 | General µP | Interlocked HS | |

^{*}B side only. See table below.



Preliminary

Features

- Asynchronous, bidirectional first-in, first-out buffer.
- Extends depth of Z-FIO without limit.
- 128 x 8 organization.

- 3-state data outputs.
- Empty and Full status pins are wire-ORed among multiple stages.

Description

The Z-FIFO first-in, first-out buffer unit is a 128 x 8-bit memory with bidirectional data transfer capability and handshake logic. Its structure is similar to that of other FIFOs that are commonly available, such as the AM2812 and the 3351. The handshake logic used is compatible with that of the Z8, the Z-CIO, and Z-FIO. Z-FIFO buffers can be cascaded, end to end, without limit, their RFD/ $\overline{\rm DAV}$ and $\overline{\rm ACKIN}$ signals daisy-chained, to make a FIFO array any desired number of words deep. Two such channels in parallel, suitably controlled, make up a 16-bit-wide buffer array.

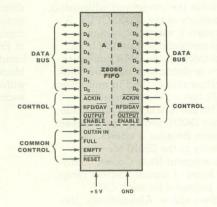


Figure 1. Pin Functions

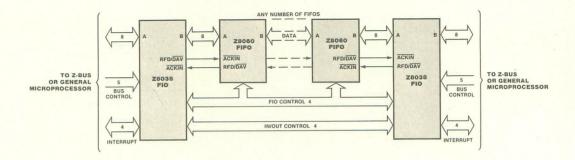


Figure 2. Using FIFOs to Extend FIOs

More to Come

The components described in the foregoing Product Briefs exist now or are well along in their development. These components represent the first step of an ongoing commitment to support the Z8000 Family. The next step is a series of specialized processors and peripherals that includes the Z8016 DMA Controller, Z8052 CRT Controller and Z8065 Burst Error Processor.

Z8016 DMA Controller. This high-speed (2M byte/sec) versatile dual-channel DMA controller matches the power and addressing capability of the Z8000 CPUs. It supports a variety of system implementations, ranging from dedicated single-DMA configurations to distributed multiple-DMA configurations found in multi-user, multi-tasking environments.

The Z8016 takes full advantage of the Z8000 memory management scheme because it interfaces directly to the Z8010 Memory Management Unit. Consequently, 8M bytes of logical address range are provided for each CPU address space. Alternatively, the Z8016 can operate independently of the Z8010 MMU and directly address up to 16M bytes of physical address space.

The ability to self-load control parameters from memory enables chained DMA operations of different types, and provides a high degree of independence from the CPU. Memory-to-Memory data transfers as well as the more conventional data transfer between I/O and memory can be executed. Data transfers can be in the form of single byte, double byte or word, and a number of search and match operations can be performed on the data.

Several different interrupt stimuli can be enabled under program control. These include interrupts on Terminal Count (TC), End of Process (EOP), or a number or condition bits in the Channel Mode Register. The Z8016 operates within the Z8000 daisy-chain vectored-priority interrupt scheme.

Z8052 CRT Controller. The Z8052 is a broad-application, raster scan CRT controller specifically designed to interface the Z8000 CPU to a variety of CRT displays.

Its numerous advanced features suit a wide variety of applications including general-business and scientific data processing, word processing and graphics.

Text-editing software implementation is simplified by means of variable vertical and horizontal split-screen capability. Double character-cell display with vertical adjustment of character position enhances word processing as well as scientific and mathematical notation. Character justification along with mixing of fonts is achieved by external synchronization of vertical retrace and variable character clock frequency. Typewriter formatted single- or multiple-line spacing along with oversized alphanumerics or memory bit-mapped displays can be generated with variable scan lines per character row. Simple line drawing capability is also made available with the line attributes provided.

The Z8052 programmable register architecture allows easy operating configuration and mode changes under software program control. This contrasts favorably to the time consuming and costly ROM programming methods used by other CRT controllers.

These advanced display features are achieved with the ability to change attributes dynamically on a real-time basis. The ability to change row attributes in real time allows smooth scrolling. Character attributes changing in real time allow multiple cursors and various underline combinations, all with individual blink rates. The character attributes control normal or double character cells for subscripted and superscripted display as well as the line attributes used for line drawing. This allows intermixing of forms and text displays for various applications.

Dynamic change of attributes is achieved by using a separate display refresh RAM in which the data for changing attributes is stored in parallel with the displayed character data. Data transfers from host processor to display RAM are made without memory access contention problems between Z8052 CRTC and processor. This is achieved by setting up the display RAM addressing so the first half of the character clock period is used for CRTC functions and the second half is used for host processor data transfer.

Z8065 Burst Error Processor.

The Z8065 Burst Error Processor provides high-speed error correction for Z8000 disk-based systems or other Z8000 systems in which high-speed data transfer takes place. With data transfer rates and storage size increasing in both floppy and solid disks, error rates without correction become increasingly intolerable. The Z8065 complements the Z8000 and its peripherals in systems using high-speed or high-density peripheral interfaces such as solid disk or floppy disk drives.

The Z8065 provides a selection of the four generator polynomials popularized by IBM, Digital Equipment Corporation and Calcomp. The divide mode performs division of the data stream by the selected polynomial to generate the check bits during data transfer to disk. For read operations, the divide mode can detect errors that may have occurred in a read or write. To locate the error for correction. a second compute mode identifies the error pattern and assists in finding the error location in the data stream.

Zeros that into Processor The 28065 Barst Error chromate provides high speed error course, tion for 28000 disk-based argumation for other 23000 disk-based argumation of other 23000 argumations in which lighters with a place. With data transfer takes an along and solid disk, error rates floopy and solid disk, error rates unjuly mitolerable. The 28065 courselphents the 28000 and its peripherals in systems using night in peripherals in systems using night in peripherals. The 28065 powides a selection light for 28065 provides a selection of the love operation and calculated population and Calculate population and Calculate for each error by the selection of the divide stream by the selection bave occurred in a read or write mode can detect error by the selection. The read operations and a second compute mode item that may also bave occurred in a read or write mode can detect error by correction. In olid the error by correction.

In olidate the error by correction as second compute mode item in a read or write mode can detect error by correction.

In olidate arms patient and contains and data arms of the error patient and calculate and a second compute mode item in the error by correction.

Dynamic change of attributes is acquieved by using a separate display refresh RAM in which the data for changing attributes is stored in parallel with the displayed character data. Data transfers from host processor to display RAM are made without memory access contention problems between 28053 CRTC and processor. This is achieved by setting up the display RAM addressing so the tirst half of the character clock period is used for CRTC unquique and the second CRTC unquique and the second half is used for host processor data

The 28052 programmable register architecture allows easy observing configuration and mode therapes under software program occurs. This contrasts layorably to be time consuming and costly of programming methods used by other CHT controllers. I have advanced display teatures are software with the ability to the software basis. The ability to display among the ability to a fillow automatical file and time relative among the ability and the ability and the ability and the among a changing in real time attributes changing in real time attributes changing in real time advance of an ability and advance of the control of the change of the change of the control of the change of the change

Z8000 Data Manual

Z8000 Data Manual

-First Edition-

定価3,200円

発行所 Zilog

10340 Bubb Road, Cupertino, California 95014 Telephone: (408) 446-4666

発売元 CQ出版株式会社

東京都豊島区巣鴨 1 −14− 2 170 170 1803 (947) 6311(代表) 振替 東京 0−10665



